TURING 图灵程序设计丛书

High Performance Browser Networking



Web性能权威指南

[加] *Ilya Grigorik* 著
李松峰 译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

李松峰

2006年起投身翻译，出版过译著30余部，包括《JavaScript高级程序设计》、《简约至上》等畅销书。2008年进入出版业，从事技术图书策划、编辑和审稿工作。

2007年创立知识分享网站“为之漫笔”（cn-cuckoo.com），翻译了大量国外经典技术文章。2012年下半年创立“A List Apart中文版”站点（alistapart.cn），旨在向中文读者译介这一国际顶级Web设计与开发杂志。他经常参加技术社区活动，曾在W3ctech 2012 Mobile上分享“Dive into Responsive Web Design”。2013年1月应邀在金山网络分享“响应式Web设计”，2013年3月应邀在奇虎360分享“JS的国”。

图灵程序设计丛书

Web性能权威指南

High Performance Browser Networking

［美］Ilya Grigorik 著
李松峰 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

Web性能权威指南 / (加) 格里高利克
(Grigorik,I.) 著 ; 李松峰译. -- 北京 : 人民邮电出
版社, 2014.4
（图灵程序设计丛书）
书名原文: High performance browser networking
ISBN 978-7-115-34910-1

Ⅰ. ①W… Ⅱ. ①格… ②李… Ⅲ. ①计算机网络—程
序设计 Ⅳ. ①TP393.09

中国版本图书馆CIP数据核字(2014)第043538号

内 容 提 要

本书是谷歌公司高性能团队核心成员的权威之作，堪称实战经验与规范解读完美结合的产物。本书目标是涵盖 Web 开发者技术体系中应该掌握的所有网络及性能优化知识。全书以性能优化为主线，从 TCP、UDP 和 TLS 协议讲起，解释了如何针对这几种协议和基础设施来优化应用。然后深入探讨了无线和移动网络的工作机制。最后，揭示了 HTTP 协议的底层细节，同时详细介绍了 HTTP 2.0、XHR、SSE、WebSocket、WebRTC 和 DataChannel 等现代浏览器新增的具有革命性的新能力。

本书适合所有 Web 应用及站点开发人员阅读，包括但不限于前端、后端、运维、大数据分析、UI/UX、存储、视频、实时消息，以及性能工程师。

◆ 著 [加] Ilya Grigorik
译 李松峰
责任编辑 李 瑛
执行编辑 李 静
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：787×1092 1/16
印张：21
字数：400千字 2014年 4 月第 1 版
印数：1－4 000册 2014年 4 月北京第 1次印刷
著作权合同登记号 图字：01-2013-7651号

定价：69.00元
读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创 新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分 无线网络性能

第三部分 HTTP

第四部分　浏览器 API 与协议

Steve Souders推荐序

“合格的开发者知道怎么做，而优秀的开发者知道为什么那么做。”

相信每一位读者看完这句话，一定打心眼儿里赞同。我们都希望自己能够理解身边的各种系统，同时还能跟别人讲得明白。然而，如果你是一名 Web 开发者，那很可能距离这个目标会越来越远。

Web 开发的分工越来越细。你在做哪一类 Web 开发？前端？后端？运维？大数据分析？ UI/UX ？存储 ？视频？实时消息？我还想再加上一个角色——性能工程师。

钻研基础知识与紧跟最新动向本身是一对矛盾，很难平衡。可是，没有基础，那只能是“墙上芦苇，头重脚轻根底浅”。光知道表面上的那点东西可不行。需要解决难题时，发生异常状况时，理解基础知识的人会脱颖而出。

正因为如此，我才说这本书非常重要，不能不看。如果你搞的是 Web 开发，那你技术体系的根基就是 Web 和它赖以存在的大量网络协议：TCP、TLS、UDP、HTTP，等等。这些协议分别有各自的性能特点和优化技巧，为开发高性能应用，你必须理解为什么网络那么运行。

说实话，我真为想读这本书的你感到庆幸！要是我刚刚接触 Web 编程时有这样一本书就好了。那样，就会有一位真正理解网络的人为我释疑解惑，告诉我那些标准和规范的要点，填充我技术体系中的空白。这本书的作者 Ilya Grigorik，是少见的网络编程专家，而本书堪称实战经验与规范解读完美结合的产物。

本书中，作者解释了网络编程中的很多为什么：为什么延迟是性能瓶颈？为什么 TCP 并不总是最优传输机制，而 UDP 有时候反而是更好的选择？为什么重用连接是关键性的优化策略？然后，他又更进一步，给出改进网络性能的具体建议。想要降低延迟？在靠近客户端的服务器上完成会话。想要提高连接重用率？保持连接持

久化。正是这种提出问题、分析问题和解决问题的模式，让本书内容极为贴近实战，接地气。

除了全面探讨网络的基础知识，作者还详细讲解了协议和浏览器的最新进展。讲了 HTTP 2.0 的诸多优点，回顾了 XHR 及其催生 CORS（Cross-Origin Resource Sharing，跨源资源共享）的局限性，还有 SSE（Server-Sent Events，服务器发送事件）、WebSockets 和 WebRTC。让我们彻底跟上了浏览器网络技术栈的最新进展。

从性能角度分析，基础和最新进展是本书特色，也是本书贯穿始终的主线。正是性能这个视角，让我们理解了网络开发中的那么多为什么，明白了这些东西怎么影响我们的网站和用户。本书把抽象的规范变成了可操作的建议，让我们可以马上学以致用去优化网站，去创造最佳用户体验。这才是最重要的。所以，一定不要错过这本书！

——Steve Souders

世界级 Web 性能专家、谷歌公司高性能工程师

《高性能网站建设指南》等畅销书作者，2013[1]

注 1：2014 年 3 月，Steve Souders 加入国际知名云加速平台 Fastly，任首席性能官。——编者注

前言

对所有程序员而言，Web 浏览器是今天部署最为广泛的应用平台。每一部智能手机，每一台平板、笔记本和桌面电脑，以及介于这些之间的其他设备，都安装了 Web 浏览器。据预测，到 2020 年，能上网的设备将突破 200 亿部，其中每一部都会安装浏览器，而且最低限度都会支持 Wi-Fi 或蜂窝连接。至于运行的是什么平台，设备是谁制造的，操作系统是哪个版本，都无所谓，反正每部设备都会带一个浏览器。就是这个浏览器，它的功能正变得越来越强大。

回首从前，那时的浏览器与现在我们使用的浏览器完全不能同日而语。浏览器革命还是最近几年的事：HTML 和 CSS 构成表现层，JavaScript 则成为 Web 上的新“汇编”语言，而新的 HTML5 API 仍在不断改进，致力于为交付吸引人的高性能应用提供新功能。可以说，有史以来还没有哪项技术或平台，在部署或装机率方面能和今天的浏览器相提并论。所以，这里有无限的机会，创新也无处不在。

而且，浏览器中网络基础设施的快速发展与创新，同样是天下无敌。过去，我们能实现的交互仅限于简单的 HTTP 请求和响应；如今，高效的流式传输、双向实时通信，以及交付自定义协议，甚至端到端视频会议和两端之间直接交换数据，都已经成为现实。而且，所有这些只不过是几十行 JavaScript 代码的事儿。

然后呢？几十亿设备互联，已有和新在线服务吸引的用户越来越多，只有高性能 Web 应用才谈得上竞争力。速度是关键！事实上，对某些应用来说，速度决定命运。要开发出高性能的 Web 应用，必须透彻理解浏览器及其网络交互机制，而这正是本书的主题。

关于本书

本书目标是涵盖开发者应该掌握的所有网络知识：网络开发中要用到哪些协议，这些协议有什么固有的局限性，如何针对底层网络优化自己的应用，浏览器提供了哪些网

络相关的功能，以及什么时候需要用到它们。

我们将从 TCP、UDP 和 TLS 协议的内部工作原理讲起，向大家解释如何针对这几种协议和基础设施来优化我们的应用。然后深入地探讨无线和移动网络的工作机制，以无线电为媒介的通信可大不一样。对此，我们将围绕如何设计和架构应用，讨论它们各自的痛点所在。最后，我们再揭示 HTTP 协议的底层细节，同时详细介绍浏览器新增的一些令人激动的能力：

- 即将到来的 HTTP 2.0 的诸多改进；
- XHR 的新特性和新功能；
- 通过 SSE 发送数据流；
- 通过 WebSocket 实现双向通信；
- 通过 WebRTC 实现端到端的音频和视频通信；
- 通过 DataChannel 实现端到端的数据交换。

要设计和开发高性能的应用，必须理解每一位数据是如何交付的，必须理解每一种传输机制和相关协议的特点。毕竟，等待网络是我们应用最大的性能瓶颈，再怎么优化渲染 JavaScript 或其他方面，也抵不上网络优化！本书的目标就是告诉读者怎么消除等待时间，利用现有网络实现最大的性能优化。

本书全面介绍了 Web 性能优化的知识和技术，适合对构建和交付高性能应用感兴趣的所有读者。简单地说，如果你不满足于那些枯燥的检查表，而更希望知晓浏览器乃至底层协议的真实工作过程，就应该读一读这本书。本书既会对配置和架构给出实用建议，也会探讨为达成优化目标而必须考虑的因素和权衡的要点，既讲“怎么办”，也讲“为什么”。

> 本书重点讨论与浏览器应用相关的各种协议及特性。不过，关于 TCP、UDP、TLS、HTTP，乃至其他每一种协议的讨论，同样也适用于本地应用，而且不局限于任何平台。

排版约定

本书使用的排版约定如下。

- 楷体
  表示新的术语。

- 等宽字体
  表示程序片段，也用于在正文中表示程序中使用的变量、函数名、命令行代码、环境变量、语句和关键词等代码文本。

- **加粗的等宽字体**
  表示应该由用户逐字输入的命令或者其他文本。

- *倾斜的等宽字体*
  表示应该由用户输入的值或根据上下文决定的值替换的文本。

这个图标代表小窍门、建议或说明。

这个图标代表警告信息。

Safari® Books Online

Safari Books Online（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920028084.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

第一部分

网络技术概览

第 1 章

延迟与带宽

1.1 速度是关键

近几年来，WPO（Web Performance Optimization，Web 性能优化）产业从无到有，快速增长，充分说明用户越来越重视速度方面的用户体验。而且，在我们这个节奏越来越快、联系越来越紧密的世界，追求速度不仅仅是一种心理上的需要，更是一种由现实事例驱动的用户需求。很多在线公司的业绩已经证实：

- 网站越快，用户的黏性越高；
- 网站越快，用户忠诚度更高；
- 网站越快，用户转化率越高。

简言之，速度是关键。要提高速度，必须先了解与之相关的各种因素，以及根本性的限制。本章主要介绍对所有网络通信都有决定性影响的两个方面：延迟和带宽（图 1-1）。

- 延迟

  分组从信息源发送到目的地所需的时间。

- 带宽

  逻辑或物理通信路径最大的吞吐量。

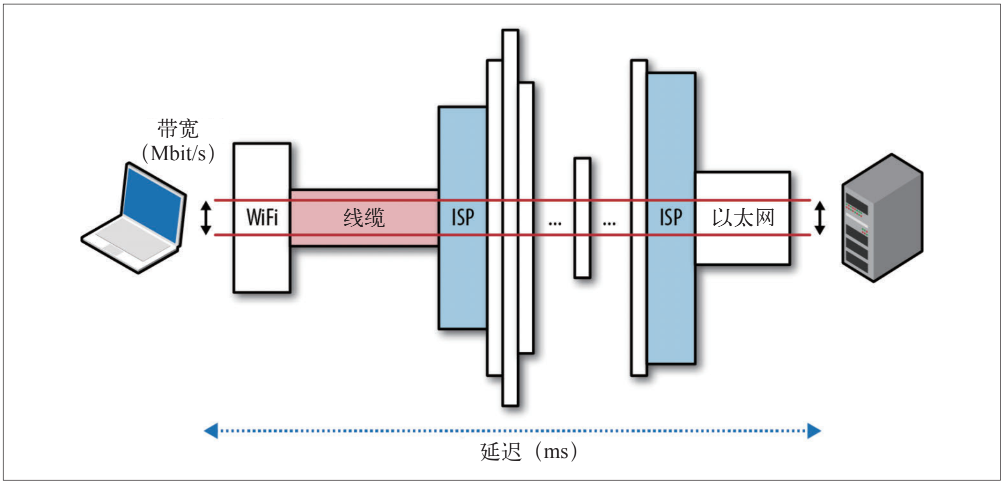


图 1-1：延迟和带宽

理解了带宽和延迟之间的关系，接下来就可以进一步探讨 TCP、UDP，以及构建于它们之上的所有应用层协议的内部构造和性能特征。

为减少跨大西洋的延迟而铺设 Hibernia Express 专线

在金融市场上，很多常用交易算法首要的考虑因素就是延迟，因为几 ms 的差距可能导致数百万美元的收益或损失。

2011 年初，华为与 Hibernia Atlantic 开始合作铺设一条横跨大西洋，连接伦敦和纽约的近 5000 km 的海底光缆（Hibernia Express）。铺设这条海底光缆的唯一目的，就是减少城市间的路由，（相对于使用其他横跨大西洋的线路）为交易商节省 5 ms 的延迟。开通运营后，这条光缆将只由金融机构使用，耗资预计达 4 亿美元。

简单计算一下，不难得出节省 1 ms 的成本是 8000 万美元。延迟的代价由此可见一斑。

1.2　延迟的构成

延迟是消息（message）或分组（packet）从起点到终点经历的时间。这个定义简单明了，但却掩盖了很多有用的信息。事实上，任何系统都有很多因素可能影响传送消息的时间。因此，弄清楚这些因素是什么，以及它们如何影响性能是最重要的。

下面看看路由器这个负责在客户端和服务器之间转发消息的设备，会牵涉哪些影响延迟的因素。

- 传播延迟
  消息从发送端到接收端需要的时间，是信号传播距离和速度的函数
- 传输延迟
  把消息中的所有比特转移到链路中需要的时间，是消息长度和链路速率的函数
- 处理延迟
  处理分组首部、检查位错误及确定分组目标所需的时间
- 排队延迟
  到来的分组排队等待处理的时间

以上延迟的时间总和，就是客户端到服务器的总延迟时间。传播时间取决于距离和信号通过的媒介，另外传播速度通常不超过光速。而传输延迟由传输链路的速率决定，与客户端到服务器的距离无关。举个例子，假设有一个 10 MB 的文件，分别通过两个链路传输，一个 1 Mbit/s，另一个 100 Mbit/s。在 1 Mbit/s 的链路上，需要花 10 s，而在 100 Mbit/s 的链路上，只需 0.1 s。

接着，分组到达路由器。路由器必须检测分组的首部，以确定出站路由，并且还可能对数据进行检查，这些都要花时间。由于这些检查通常由硬件完成，因此相应的延迟一般非常短，但再短也还是存在。最后，如果分组到达的速度超过了路由器的处理能力，那么分组就要在入站缓冲区排队。数据在缓冲区排队等待的时间，当然就是排队延迟。

每个分组在通过网络时都会遇到这样或那样的延迟。发送端与接收端的距离越远，传播时间就越长。一路上经过的路由器越多，每个分组的处理和传输延迟就越多。最后，网络流量越拥挤，分组在入站缓冲区中被延迟的可能性就越大。

本地路由器的缓冲区爆满

缓冲区爆满（Bufferbloat）是 Jim Gettys 在 2010 年发明的一个术语，是排队延迟影响网络整体性能的一个形象的说法。

造成这个问题的原因主要是如今市面上的路由器都会配备很大的入站缓冲区，以便“不惜一切代价”避免丢包（分组）。可是，这种做法破坏了 TCP 的拥塞预防机制（congestion avoidance，下一章将介绍），导致网络中产生较长且可变的延迟时间。

为解决这个问题，有人提出了新的 CoDel 主动队列管理算法，且已经在 Linux 内核 3.5 以上版本中实现。如果想了解更多内容，可以参考 ACM 的一篇论文，搜索“Controlling Queue Delay”就能找到。

1.3　光速与传播延迟

正如爱因斯坦在他的狭义相对论里所说的，光速是所有能量、物质和信息运动所能达到的最高速度。这个结论给网络分组的传播速度设定了上限。

好消息是光速极快，每秒能达到 299 792 458 米（大约 30 万公里）。但是，别忘了还有个但是，这是光在真空中的传播速度。而网络中的分组是通过铜线、光纤等介质传播的，这些介质会导致传播速度变慢。光速与分组在介质中传播速度之比，叫做该介质的折射率。这个值越大，光在该介质中传播的速度就越慢。

传播分组的光纤，大多数折射率从 1.4 到 1.6 不等。不过，我们也在逐渐改进传播材料的质量，从而不断降低折射率。为简单起见，我们大都假定光通过光纤的速度约为每秒 200 000 000 米，对应的折射率约为 1.5。值得一提的是，我们已经能够把折射率降低到最大速度的一个很小的常数因子的范围内了！仅此就堪称一项了不起的成就。

当然，我们还是不太习惯以光速为参照来思考，因此表 1-1 给出了几个例子，以便我们能够直观地想象。

表1-1：真空与光纤中的信号延迟

路线	距离（km）	时间：光在真空中	时间：光在光纤中	光纤中的往返时间（RTT）
纽约到旧金山	4 148	14 ms	**21 ms**	42 ms
纽约到伦敦	5 585	19 ms	**28 ms**	56 ms
纽约到悉尼	15 993	53 ms	**80 ms**	160 ms
赤道周长	40 075	133.7 ms	**200 ms**	400 ms

光速已经很快了，尽管如此从纽约到悉尼的一个往返（RTT）也要花 160 ms。事实上，以上这些数字都是理想情况下的结果，因为我们假设传送分组的光缆恰好是连接两个城市的一条完美的大弧形线路（地球表面两点间最短的距离）。而实际上纽约和悉尼之间是没有这样一条线路的，分组旅行的距离比这要长得多。这条线路中的每一跳都会涉及寻路、处理、排队和传输延迟。结果呢，纽约到悉尼的实际 RTT，大约在 200~300 ms 之间。即便如此，还是很快的，对吧？

我们都不习惯用 ms 来度量身边的事物，但研究表明：在软件交互中，哪怕 100~200 ms 左右的延迟，我们中的大多数人就会感觉到“拖拉”；如果超过了 300 ms 的门槛，那就会说“反应迟钝”；而要是延迟达到 1000 ms（1s）这个界限，很多用户就会在等待响应的时候分神，有人会想入非非，有人恨不得忙点别的什么事儿。

结论很简单：要想给用户最佳的体验，而且保证他们全神贯注于手边的任务，我们

的应用必须在几百 ms 之内响应。这几乎没有给我们——特别是网络，留出多少出错的余地。若要成功，必须认真对待网络延迟，在每个开发阶段都为它设立明确的标准。

> CDN（Content Delivery Network，内容分发网络）服务的用途很多，但最重要的就是通过把内容部署在全球各地，让用户从最近的服务器加载内容，大幅降低传播分组的时间。
>
> 或许我们不能让数据传输得更快，但我们可以缩短服务器与用户之间的距离！把数据托管到 CDN 能够显著提高性能。

1.4 延迟的最后一公里

你说怪不怪，延迟中相当大的一部分往往花在了最后几公里，而不是在横跨大洋或大陆时产生的，这就是所谓的“最后一公里”问题。为了让你家或你的办公室接入互联网，本地 ISP 需要在附近安装多个路由收集信号，然后再将信号转发到本地的路由节点。连接类型、路由技术和部署方法五花八门，分组传输中的这前几跳往往要花数十 ms 时间才能到达 ISP 的主路由器！根据美国联邦通信委员会（FCC）发布于 2012 年年中的《美国宽带测量报告》（*Measuring Broadband America*），在通信高峰的几个小时内：

> 光纤入户服务的平均往返时间为 18 ms，有线电视线路上网平均为 26 ms，DSL 专线平均为 43 ms。
>
> ——FCC，2012 年 7 月

这里 18~43 ms 的延迟测量的还只是 ISP 核心网络中与用户最近的节点，此时分组甚至都还没有启程呢！ FCC 的报告只反映了美国的情况，但最后一公里的延迟却是世界任何一个角落的互联网提供商共同面临的问题。如果你好奇，那只要一条简单的 `traceroute` 命令，就能知道上网服务商的拓扑结构和速度。

```
$> traceroute google.com
traceroute to google.com (74.125.224.102), 64 hops max, 52 byte packets
 1  10.1.10.1 (10.1.10.1) 7.120 ms 8.925 ms 1.199 ms ❶
 2  96.157.100.1 (96.157.100.1) 20.894 ms 32.138 ms 28.928 ms
 3  x.santaclara.xxxx.com (68.85.191.29) 9.953 ms 11.359 ms 9.686 ms
 4  x.oakland.xxx.com (68.86.143.98) 24.013 ms 21.423 ms 19.594 ms
 5  68.86.91.205 (68.86.91.205)  16.578 ms 71.938 ms 36.496 ms
 6  x.sanjose.ca.xxx.com (68.86.85.78) 17.135 ms 17.978 ms 22.870 ms
 7  x.529bryant.xxx.com (68.86.87.142) 25.568 ms 22.865 ms 23.392 ms
 8  66.208.228.226 (66.208.228.226) 40.582 ms 16.058 ms 15.629 ms
 9  72.14.232.136 (72.14.232.136) 20.149 ms 20.210 ms 18.020 ms
10  64.233.174.109 (64.233.174.109) 63.946 ms 18.995 ms 18.150 ms
11  x.1e100.net (74.125.224.102) 18.467 ms 17.839 ms 17.958 ms ❷
```

❶ 第 1 跳：本地无线路由器
❷ 第 11 跳：谷歌服务器

分组从森尼维耳市开始，跳到圣克拉拉，经过奥克兰，返回圣何塞，又被路由到“529 Bryant”数据中心，从那儿才开始向谷歌服务器进发，最终在第 11 跳到达目的地。整个行程大约 18 ms，所有延迟都算上了，还不错。但与此同时，我们的分组几乎穿越了大半个美国本土！

最后一公里的延迟与提供商、部署方法、网络拓扑，甚至一天中的哪个时段都有很大关系。作为最终用户，如果你想提高自己上网的速度，那选择延迟最短的 ISP 是最关键的。

> 大多数网站性能的瓶颈都是延迟，而不是带宽！要理解为什么，需要明白 TCP 和 HTTP 协议的细节，这也是本书后面几章要讨论的。假如你现在就着急知道，可以直接翻到 10.3.1 节“更多带宽其实不（太）重要”。

使用 traceroute 测量延迟

traceroute 是一个简单的网络诊断工具，可以列出分组经过的路由节点，以及它在 IP 网络中每一跳的延迟。为找到每一跳的节点，它会向目标发送一系列分组，每次发送时的“跳数限制”都会递增（1、2、3，等等）。在达到跳数限制时，中间的节点会返回 ICMP Time Exceeded 消息，traceroute 根据这个消息可以计算出每一跳的延迟。

在 Unix 平台上，可以在命令行运行 `traceroute`。而在 Windows 平台中，相应的命令叫 `tracert`。

1.5 网络核心的带宽

光纤就是一根“光导管”，比人的头发稍微粗一点，专门用来从一端向另一端传送光信号。金属线则用于传送电信号，但信号损失、电磁干扰较大，同时维护成本也较高。这两种线路我们的数据分组很可能都会经过，但一般长距离的分组传输都是通过光纤完成的。

通过波分复用（WDM，Wavelength-Division Multiplexing）技术，光纤可以同时传输很多不同波长（信道）的光，因而具有明显的带宽优势。一条光纤连接的总带宽，等于每个信道的数据传输速率乘以可复用的信道数。

到 2010 年初，研究人员已经可以在每个信道中耦合 400 多种波长的光线，最大容量可达 171 Gbit/s，而一条光纤的总带宽能够达到 70 Tbit/s！如此大的吞吐量如果换成铜线（传输电信号）可能需要几千条。自然地，像两个大陆间的海底数据传输，现在都已经使用光纤连接了。每条光缆会封装几条光纤（常见的是 4 条），折算出来的带宽容量能达到每秒几百太比特。

1.6 网络边缘的带宽

构成因特网核心数据路径的骨干或光纤连接，每秒能够移动数百太比特信息。然而，网络边缘的容量就小得多了，而且很大程度上取决于部署技术，比如拔号连接、DSL、电缆、各种无线技术、光纤到户，甚至与局域网路由器的性能也有关系。用户可用带宽取决于客户端与目标服务器间最低容量连接（参见图 1-1）。

Akamai 技术公司在全球部署了 CDN，服务器遍及世界各地，而且每季度都会发布一份免费的带宽平均速度报告（由他们的服务器测量），地址为：http://www.akamai.io。表 1-2 中展示的是 2012 年年中监测到的世界各地的平均带宽。

表1-2：2012年年中Akamai公司服务器的平均带宽

排名	国家和地区	平均Mbit/s	年增长
-	全球	**3.1**	17%
1	韩国	**14.2**	−10%
2	日本	**11.7**	6.8%
3	香港	**10.9**	16%
4	瑞士	**10.1**	24%
5	荷兰	**9.9**	12%
…			
9	美国	**8.6**	27%

以上数据不包括移动网络的流量，这一块我们稍后还要详细讨论。目前，可以肯定移动网络的速度差异很大，而且一般都更慢。即便如此，2012 年上半年全球宽带的带宽也只有 2.6 Mbit/s！韩国以 15.7 Mbit/s 的平均带宽位居第 1，美国的 6.7 Mbit/s 位列第 12 名。

为了更好地理解这些数字，我们举个例子：视频网站的高清视频流，根据分辨率高低以及编解码器不同，可能需要 2~10 Mbit/s。因此，一般用户在网络末端观看低分辨率的流视频几乎就可以消耗掉其所有带宽。对于一个可能会有多人同时上网的家庭而言，这个结果并不理想。

搞清楚每个用户的带宽瓶颈通常不是件容易的事，却又非常重要。同样，为了满足大家的好奇心，推荐一个在线服务吧：Ookla 运营的 http://speedtest.net（图 1-2），可以测试客户端到某个本地服务器的上传和下载速度。在后面讨论 TCP 的时候，我们会解释为什么选择本地服务器很重要。在这些服务器上运行测试可以验证你的 ISP 在广告中吹嘘的速度。

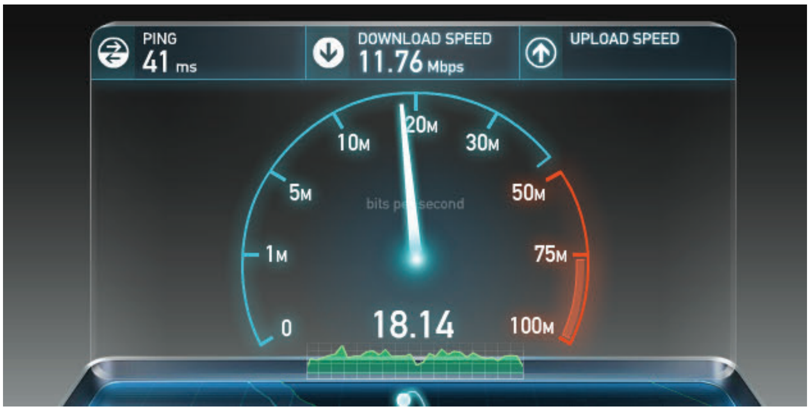


图 1-2：上传和下载速度测试（speedtest.net）

虽然与 ISP 的高带宽连接是必要的，但这个高带宽并能不保证端到端的传输速度。由于请求密集、硬件故障、网络攻击，以及其他很多原因，网络的某个中间节点随时都有可能发生拥塞。吞吐量和延迟波动大是因特网固有的特点。要预见、控制、适应瞬息万变的“网络天气”可不容易。

1.7 目标：高带宽和低延迟

人们对高带宽的需求增长迅速，很大程度是受到了在线流视频的拉动，目前的视频流量已经占到全部因特网流量的一半以上。好在，虽然不一定很便宜，但我们有很多方法可以提高容量。比如，可以在光纤链路中部署更多光纤、在拥塞的路由之间铺设更多线路，甚至是改进 WDM 技术，以便让现有连接能够传输更多数据。

电信市场研究及咨询公司 TeleGeography 估计，我们到 2011 年（平均）只使用了海底光缆可用容量的 20% 左右。更重要的是，2007 年到 2011 年，太平洋海底光缆全部新增容量中有一半以上是因为 WDM 升级带来的：光缆还是那些光缆，但两端多路传输数据的技术进步了。当然，技术进步也不是没有止境，任何介质超过一定限度都会出现性能递减效应。不管怎样，只要企业的经济条件允许，就没有理由认为带宽会停止增长的脚步。就算技术停滞不前，还是可以铺设更多的光缆。

另一方面，减少延迟时间则要困难得多。通过提升光纤线路的质量，可以让光信号传输的速度更接近光速，比如采用折射率更低的材料、速度更快的路由器和中继器。然而，当前光纤折射率已经达到了 1.5 左右，最大的提升幅度预计在 30% 左右。

反过来想，不能让光线跑得更快，但可以把距离缩短。地球上两点之间的最短距离，取决于这两点之间的大圆弧。事实上，我们在设计和铺设电缆时，都是尽量缩短距离的。当然，考虑到地形特点、社会政治原因，以及相关成本，有些线路也不是最短的。总之，光速为减少延迟设定了上限，而从很多方面来看，我们的基础设施似乎也已经达到了这个极限。

遗憾的是，人类不太可能跳出物理定律的“掌心”。如果需要针对延迟采取优化措施，就必须从设计和优化协议及应用着手，并且时刻牢记光速的限制。可以减少往返、把数据部署到接近客户端的地方，以及在开发应用时通过各种技术隐藏延迟。

第 2 章

TCP的构成

因特网有两个核心协议：IP 和 TCP。IP，即 Internet Protocol（因特网协议），负责联网主机之间的路由选择和寻址；TCP，即 Transmission Control Protocol（传输控制协议），负责在不可靠的传输信道之上提供可靠的抽象层。TCP/IP 也常被称为“因特网协议套件”（Internet Protocol Suite），是由 Vint Cerf 和 Bob Khan 在他们 1974 的论文“A Protocol for Packet Network Intercommunication”（一种分组网络互通的协议）中首次提出来的。

最早的建议（RFC 675）经过几次修订，于 1981 年作为 TCP/IP 标准第 4 版发布。发布时并不是一个标准，而是两个独立的 RFC：

- RFC 791 —— Internet Protocol；
- RFC 793 —— Transmission Control Protocol。

从那时起，TCP 经过了多次改进和完善，但核心内容变化不大。TCP 很快取代了之前的协议，成为 World Wide Web、文件传输、P2P 等众多流行应用的选择。

TCP 负责在不可靠的传输信道之上提供可靠的抽象层，向应用层隐藏了大多数网络通信的复杂细节，比如丢包重发、按序发送、拥塞控制及避免、数据完整，等等。采用 TCP 数据流可以确保发送的所有字节能够完整地被接收到，而且到达客户端的顺序也一样。也就是说，TCP 专门为精确传送做了优化，但并未过多顾及时间。正如稍后我们会谈到的，这一点也给优化浏览器 Web 性能带来了挑战。

HTTP 标准并未规定 TCP 就是唯一的传输协议。如果你愿意，还可以通过 UDP（用

户数据报协议）或者其他可用协议来发送 HTTP 消息。但在现实当中，由于 TCP 提供了很多有用的功能，几乎所有 HTTP 流量都是通过 TCP 传送的。

因此，理解 TCP 的某些核心机制就成为了优化 Web 体验的必修课。虽然我们一般不会直接使用 TCP 套接口，但应用层的一些决定可能会对 TCP 以及底层网络的性能产生极大影响。

> **TCP 和 IP 协议的历史**
>
> 我们都知道有 IPv4 和 IPv6，那 IPv1~3 和 IPv5 呢？ IPv4 中的 4 表示 TCP/IP 协议的第 4 个版本，发布于 1981 年 9 月。最初的 TCP/IP 建议中同时包含两个协议，但标准草案第 4 版将这两个协议分开，使之各自成为独立的 RFC。实际上，IPv4 中的 v4 只是表明了它与 TCP 前 3 个版本的承继关系，之前并没有单独的 IPv1、IPv2 或 IPv3 协议。
>
> 1994 年，当工作组着手制定 Internet Protocol next generation（IPng）需要一个新版本号时，v5 已经被分配给了另一个试验性协议 Internet Stream Protocol（ST）。但 ST 一直没有什么进展，这也是我们为什么很少听说它的原因。结果 TCP/IP 的下一版本就成了 IPv6。

2.1 三次握手

所有 TCP 连接一开始都要经过三次握手（见图 2-1）。客户端与服务器在交换应用数据之前，必须就起始分组序列号，以及其他一些连接相关的细节达成一致。出于安全考虑，序列号由两端随机生成。

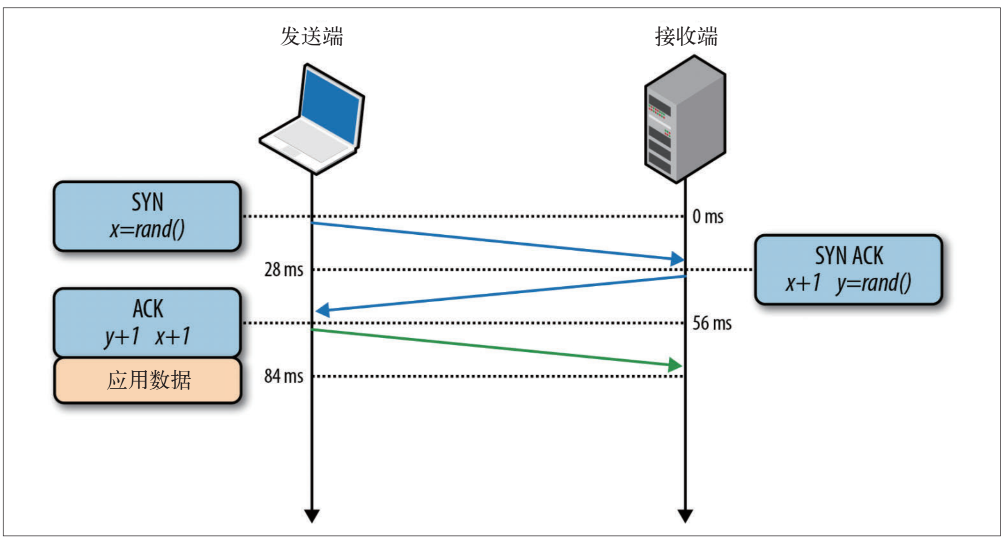


图 2-1：三次握手

- SYN

  客户端选择一个随机序列号 *x*，并发送一个 SYN 分组，其中可能还包括其他 TCP 标志和选项。

- SYN ACK

  服务器给 *x* 加 1，并选择自己的一个随机序列号 *y*，追加自己的标志和选项，然后返回响应。

- ACK

  客户端给 *x* 和 *y* 加 1 并发送握手期间的最后一个 ACK 分组。

三次握手完成后，客户端与服务器之间就可以通信了。客户端可以在发送 ACK 分组之后立即发送数据，而服务器必须等接收到 ACK 分组之后才能发送数据。这个启动通信的过程适用于所有 TCP 连接，因此对所有使用 TCP 的应用具有非常大的性能影响，因为每次传输应用数据之前，都必须经历一次完整的往返。

举个例子，如果客户端在纽约，服务器在伦敦，要通过光纤启动一次新的 TCP 连接，光三次握手至少就要花 56 ms（参见表 1-1）：向伦敦发送分组需要 28 ms，响应发回纽约又要 28 ms。在此，连接的带宽对时间没有影响，延迟完全取决于客户端和服务器之间的往返时间，这其中主要是纽约到伦敦之间的传输时间。

三次握手带来的延迟使得每创建一个新 TCP 连接都要付出很大代价。而这也决定了提高 TCP 应用性能的关键，在于想办法重用连接。

TCP 快速打开

遗憾的是，连接并不是想重用就可以重用的。事实上，由于非常短的 TCP 连接在互联网上随处可见，握手阶段已经成为影响网络总延迟的一个重要因素。为解决这个问题，人们正在积极寻找各种方案，其中 TFO（TCP Fast Open，TCP 快速打开）就是这样一种机制，它致力于减少新建 TCP 连接带来的性能损失。

经过流量分析和网络模拟，谷歌研究人员发现 TFO 平均可以降低 HTTP 事务网络延迟 15%、整个页面加载时间 10% 以上。在某些延迟很长的情况下，降低幅度甚至可达 40%。

Linux 3.7 及之后的内核已经在客户端和服务器中支持 TFO，因此成为了客户端和服务器操作系统选型的有力候选方案。即便如此，TFO 并不能解决所有问题。它虽然有助于减少三次握手的往返时间，但却只能在某些情况下有效。比如，随同 SYN 分组一起发送的数据净荷有最大尺寸限制、只能发送某些类型的 HTTP 请求，以及由于依赖加密 cookie，只能应用于重复的连接。要了解有关 TFO 容量及局限性的更多细节，请参考 IETF 最新的“TCP Fast Open”草案。

2.2 拥塞预防及控制

1984 年初，John Nagle 提到了一个被称为“拥塞崩溃”的现象，这个现象会影响节点间带宽容量不对称的任何网络：

> 拥塞控制是复杂网络中众所周知的一个问题。我们发现国防部的 Internet Protocol（IP）——纯粹的数据报协议，和 Transmission Control Protocol（TCP）——传输层协议，在一块使用时，由于传输层与数据报层之间的交互，会导致一些不常见的拥塞问题。特别是 IP 网关容易受到我们称为“拥塞崩溃”现象的严重影响，尤其是在这种网关连接不同带宽的网络时……
>
> 可能是往返时间超过了所有主机的最大中断间隔，于是相应的主机会在网络中制造越来越多的数据报副本，使得整个网络陷入瘫痪。最终，所有交换节点的缓冲区都将被填满，多出来的分组必须删掉。目前的分组往返时间已经设定为最大值。主机会把每个分组都发送好几次，结果每个分组的某个副本会抵达目标。这就是拥塞崩溃。
>
> 这种情况永远存在。达到饱和状态时，只要选择被删除分组的算法适当，网络就可以退而求其次地持续运行下去。
>
> —— John Nagle - RFC 896

这份报告的结论是拥塞崩溃不会对 ARPANET 造成影响，因为其大多数节点的带宽相同，而且其骨干网的容量相对大得多。然而，这两种情况没有持续太久。1986 年，随着加入网络的节点数量（5000+）及类型日益增多，该网络中发生了一系列拥塞崩溃故障。个别情况下，容量下降为千分之一，网络完全瘫痪。

为了解决这些问题，TCP 加入了很多机制，以便控制双向发送数据的速度，比如流量控制、拥塞控制和拥塞预防机制。

> ARPANET（Advanced Research Projects Agency Network，高级研究计划局网络）是现代互联网的前身，是世界上第一个实际运行的分组交换网络。这个项目于 1959 年正式启动，1983 年 TCP/IP 作为主要通信协议取代了原来的 NCP（Network Control Program）协议。后来呢，那大家就都知道了。

2.2.1 流量控制

流量控制是一种预防发送端过多向接收端发送数据的机制。否则，接收端可能因为忙碌、负载重或缓冲区既定而无法处理。为实现流量控制，TCP 连接的每一方都要通告（图 2-2）自己的接收窗口（rwnd），其中包含能够保存数据的缓冲区空间大小信息。

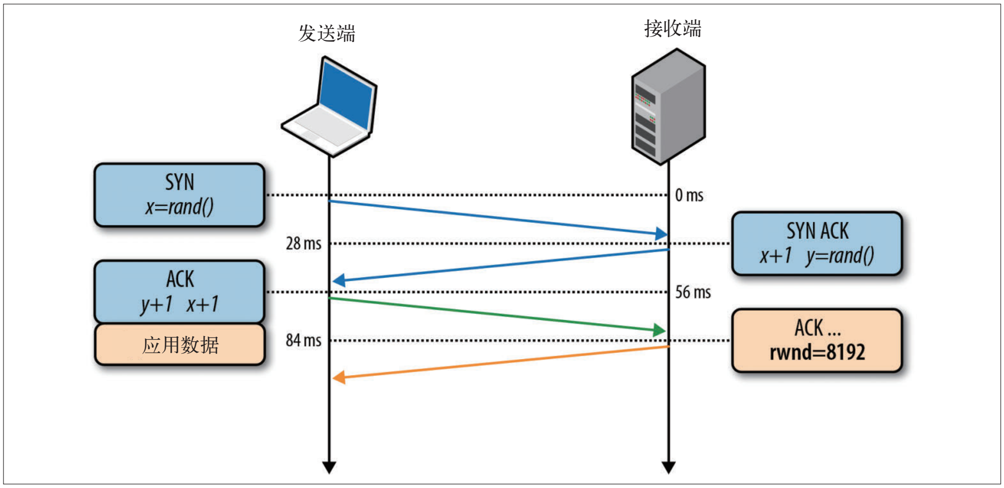


图 2-2：通告接收窗口（rwnd）大小

第一次建立连接时，两端都会使用自身系统的默认设置来发送 rwnd。浏览网页通常主要是从服务器向客户端下载数据，因此客户端窗口更可能成为瓶颈。然而，如果是在上传图片或视频，即客户端向服务器传送大量数据时，服务器的接收窗口又可能成为制约因素。

不管怎样，如果其中一端跟不上数据传输，那它可以向发送端通告一个较小的窗口。假如窗口为零，则意味着必须由应用层先清空缓冲区，才能再接收剩余数据。这个过程贯穿于每个 TCP 连接的整个生命周期：每个 ACK 分组都会携带相应的最新 rwnd 值，以便两端动态调整数据流速，使之适应发送端和接收端的容量及处理能力。

窗口缩放（RFC 1323）

最初的 TCP 规范分配给通告窗口大小的字段是 16 位的，这相当于设定了发送端和接收端窗口的最大值（216 即 65 535 字节）。结果，在这个限制内经常无法获得最优性能，特别是在那些“带宽延迟积”（参见 2.3 节“带宽延迟积”）很高的网络中。

为解决这个问题，RFC 1323 提供了“TCP 窗口缩放”（TCP Window Scaling）选项，可以把接收窗口大小由 65 535 字节提高到 1G 字节！缩放 TCP 窗口是在三次握手期间完成的，其中有一个值表示在将来的 ACK 中左移 16 位窗口字段的位数。

今天，TCP 窗口缩放机制在所有主要平台上都是默认启用的。不过，中间节点和路由器可以重写，甚至完全去掉这个选项。如果你的服务器或客户端的连接不能完全利用现有带宽，那往往该先查一查窗口大小。在 Linux 中，可以通过如下命令检查和启用窗口缩放选项：

- `$> sysctl net.ipv4.tcp_window_scaling`
- `$> sysctl -w net.ipv4.tcp_window_scaling=1`

2.2.2 慢启动

尽管 TCP 有了流量控制机制，但网络拥塞崩溃仍然在 1980 年代中后期浮出水面。流量控制确实可以防止发送端向接收端过多发送数据，但却没有机制预防任何一端向潜在网络过多发送数据。换句话说，发送端和接收端在连接建立之初，谁也不知道可用带宽是多少，因此需要一个估算机制，然后还要根据网络中不断变化的条件而动态改变速度。

要说明这种动态适应机制的好处，可以想象你在家里观看一个大型的流视频。视频服务器会尽最大努力根据你的下行连接提供最高品质信息。而此时，你家里又有人打开一个新连接下载某个软件的升级包。可供视频流使用的下行带宽一下子少了很多，视频服务器必须调整它的发送速度。否则，如果继续保持同样的速度，那么数据很快就会在某个中间的网关越积越多，最终会导致分组被删除，从而降低网络传输效率。

1988 年，Van Jacobson 和 Michael J. Karels 撰文描述了解决这个问题的几种算法：慢启动、拥塞预防、快速重发和快速恢复。这 4 种算法很快被写进了 TCP 规范。事实上，正是由于这几种算法加入 TCP，才让因特网在 20 世纪 80 年代末到 90 年代初流量暴增时免于大崩溃。

要理解慢启动，最好看一个例子。同样，假设纽约有一个客户端，尝试从位于伦敦的服务器上取得一个文件。首先，三次握手，而且在此期间双方各自通过 ACK 分组通告自己的接收窗口（rwnd）大小（图 2-2）。在发送完最后一次 ACK 分组后，就可以交换应用数据了。

此时，根据交换数据来估算客户端与服务器之间的可用带宽是唯一的方法，而且这也是慢启动算法的设计思路。首先，服务器通过 TCP 连接初始化一个新的拥塞窗口（cwnd）变量，将其值设置为一个系统设定的保守值（在 Linux 中就是 initcwnd）。

- 拥塞窗口大小（cwnd）
  发送端对从客户端接收确认（ACK）之前可以发送数据量的限制。

发送端不会通告 cwnd 变量，即发送端和接收端不会交换这个值。此时，位于伦敦的服务器只是维护这么一个私有变量。此时又有一条新规则，即客户端与服务器之间最大可以传输（未经 ACK 确认的）数据量取 rwnd 和 cwnd 变量中的最小值。那服务器和客户端怎么确定拥塞窗口大小的最优值呢？毕竟，网络状况随时都在变化，即使相同的两个网络节点之间也一样（前面的例子已经展示了这一点）。如果能通过算法来确定每个连接的窗口大小，而不用手工调整就最好了。

解决方案就是慢启动，即在分组被确认后增大窗口大小，慢慢地启动！最初，cwnd 的值只有 1 个 TCP 段。1999 年 4 月，RFC 2581 将其增加到了 4 个 TCP 段。2013 年 4 月，RFC 6928 再次将其提高到 10 个 TCP 段。

新 TCP 连接传输的最大数据量取 rwnd 和 cwnd 中的最小值，而服务器实际上可以向客户端发送 4 个 TCP 段，然后就必须停下来等待确认。此后，每收到一个 ACK，慢启动算法就会告诉服务器可以将它的 cwnd 窗口增加 1 个 TCP 段。每次收到 ACK 后，都可以多发送两个新的分组。TCP 连接的这个阶段通常被称为“指数增长”阶段（图 2-3），因为客户端和服务器都在向两者之间网络路径的有效带宽迅速靠拢。

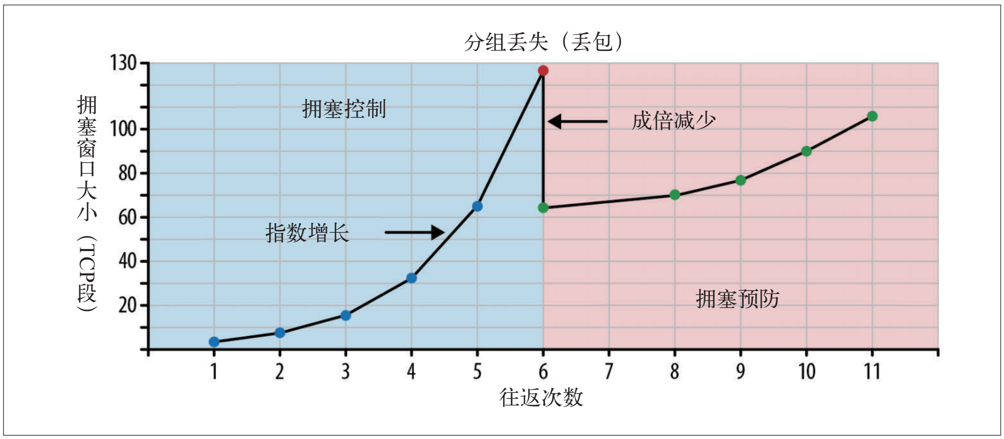


图 2-3：拥塞控制和拥塞预防

为什么知道有个慢启动对我们构建浏览器应用这么重要呢？因为包括 HTTP 在内的很多应用层协议都运行在 TCP 之上，无论带宽多大，每个 TCP 连接都必须经过慢启动阶段。换句话说，我们不可能一上来就完全利用连接的最大带宽！

相反，我们要从一个相对较小的拥塞窗口开始，每次往返都令其翻倍（指数式增长）。而达到某个目标吞吐量所需的时间，就是客户端与服务器之间的往返时间和初始拥塞窗口大小的函数（公式 2-1）。

公式 2-1：cwnd 大小达到 *N* 所需的时间

$$\text{时间} = \text{往返时间} \times \left\lceil \log_2\left(\frac{N}{\text{初始 cwnd}}\right) \right\rceil$$

下面我们就来看一个例子，假设：

- 客户端和服务器的接收窗口为 65 535 字节（64 KB）；

- 初始的拥塞窗口：4 段（RFC 2581）；
- 往返时间是 56 ms（伦敦到纽约）。

> 这里及后面例子中的初始拥塞窗口都会使用原来（RFC 2581 规定）的 4 段，因为这仍然是目前大多数服务器中常见的值。当然，你肯定不会犯这种错误的，对吧？接下来的例子能很好地说明为什么你该更新内核了。

先不管 64 KB 的接收窗口，新 TCP 连接的吞吐量一开始是受拥塞窗口初始值限制的。计算可知，要达到 64 KB 的限制，需要把拥塞窗口大小增加到 45 段，而这需要 224 ms：

$$\frac{65\,535\ \text{字节}}{1460\ \text{字节}} \approx 45\ \text{段}$$

$$56\ \text{ms} \times \left\lceil \log_2\left(\frac{45}{4}\right) \right\rceil = 224\ \text{ms}$$

要达到客户端与服务器之间 64 KB 的吞吐量，需要 4 次往返（图 2-4），几百 ms 的延迟！至于客户端与服务器之间实际的连接速率是不是在 Mbit/s 级别，丝毫不影响这个结果。这就是慢启动。

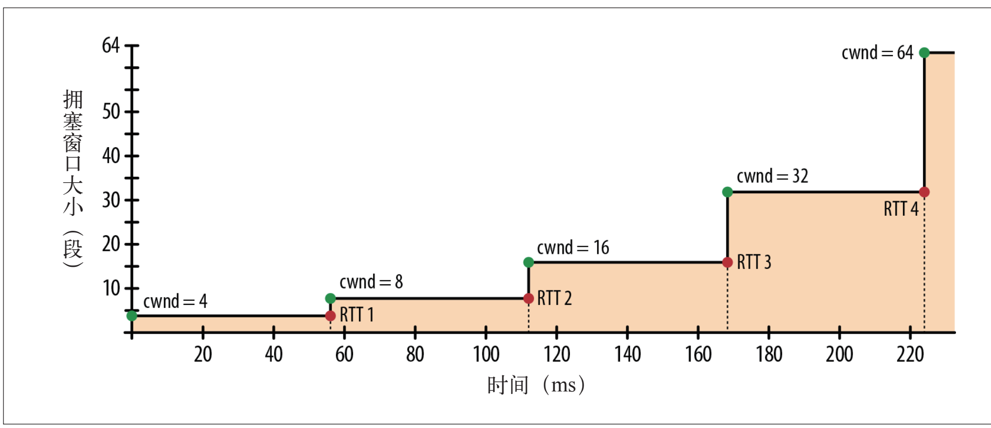


图 2-4：拥塞窗口大小增长示意图

为减少增长到拥塞窗口的时间，可以减少客户端与服务器之间的往返时间。比如，把服务器部署到地理上靠近客户端的地方。要么，就把初始拥塞窗口大小增加到 RFC 9828 规定的 10 段。

慢启动导致客户端与服务器之间经过几百 ms 才能达到接近最大速度的问题，对于大型流式下载服务的影响倒不显著，因为慢启动的时间可以分摊到整个传输周期内消化掉。

可是，对于很多 HTTP 连接，特别是一些短暂、突发的连接而言，常常会出现还没有达到最大窗口请求就被终止的情况。换句话说，很多 Web 应用的性能经常受到服务器与客户端之间往返时间的制约。因为慢启动限制了可用的吞吐量，而这对于小文件传输非常不利。

慢启动重启

除了调节新连接的传输速度，TCP 还实现了 SSR（Slow-Start Restart，慢启动重启）机制。这种机制会在连接空闲一定时间后重置连接的拥塞窗口。道理很简单，在连接空闲的同时，网络状况也可能发生了变化，为了避免拥塞，理应将拥塞窗口重置回“安全的”默认值。

毫无疑问，SSR 对于那些会出现突发空闲的长周期 TCP 连接（比如 HTTP 的 keep-alive 连接）有很大的影响。因此，我们建议在服务器上禁用 SSR。在 Linux 平台，可以通过如下命令来检查和禁用 SSR：

- `$> sysctl net.ipv4.tcp_slow_start_after_idle`
- `$> sysctl -w net.ipv4.tcp_slow_start_after_idle=0`

为演示三次握手和慢启动对简单 HTTP 传输的影响，我们假设纽约的客户端需要通过 TCP 连接向伦敦的服务器请求一个 20 KB 的文件（图 2-5），下面列出了连接的参数。

- 往返时间：56 ms。
- 客户端到服务器的带宽：5 Mbit/s。
- 客户端和服务器接收窗口：65 535 字节。
- 初始的拥塞窗口：4 段（4×1460 字节 ≈ 5.7 KB）。
- 服务器生成响应的处理时间：40 ms。
- 没有分组丢失、每个分组都要确认、GET 请求只占 1 段。

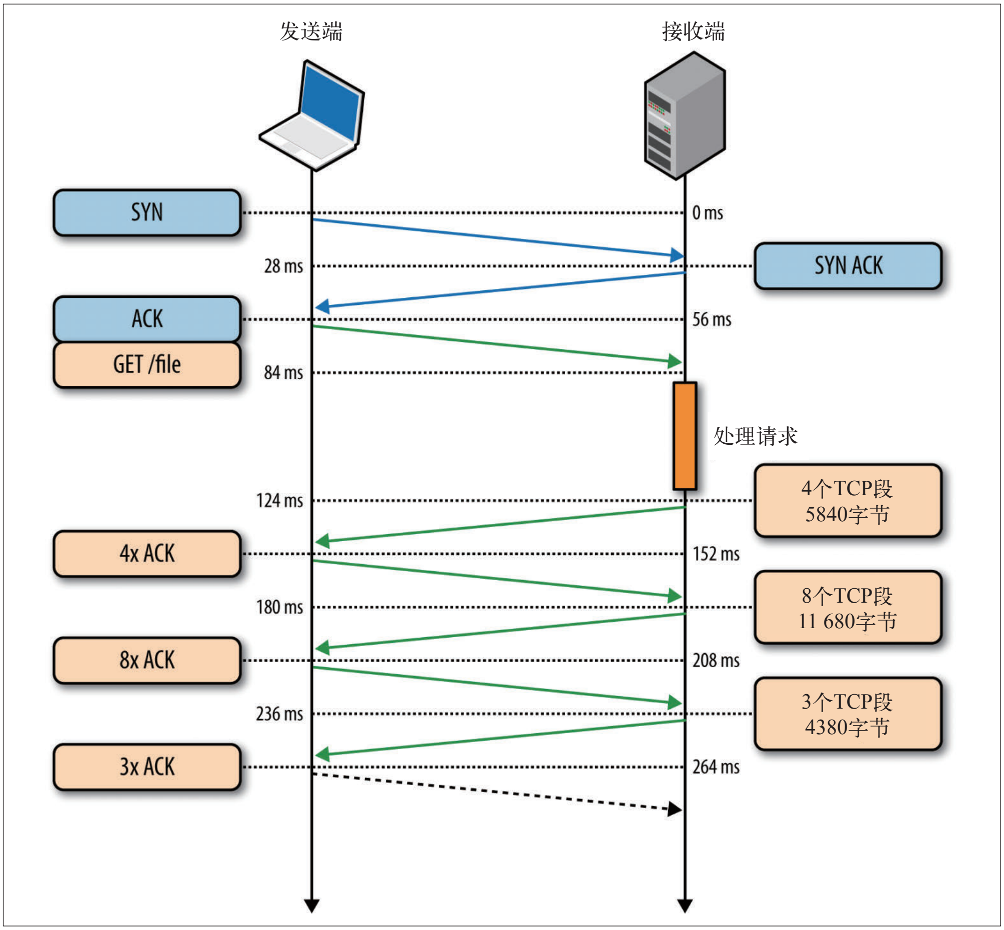


图 2-5：通过新 TCP 连接取得文件

- 0 ms：客户端发送 SYN 分组开始 TCP 握手。
- 28 ms：服务器响应 SYN-ACK 并指定其 rwnd 大小。
- 56 ms：客户端确认 SYN-ACK，指定其 rwnd 大小，并立即发送 HTTP GET 请求。
- 84 ms：服务器收到 HTTP 请求。
- 124 ms：服务器生成 20 KB 的响应，并发送 4 个 TCP 段（初始 cwnd 大小为 4），然后等待 ACK。
- 152 ms：客户端收到 4 个段，并分别发送 ACK 确认。
- 180 ms：服务器针对每个 ACK 递增 cwnd，然后发送 8 个 TCP 段。
- 208 ms：客户端接收 8 个段，并分别发送 ACK 确认。
- 236 ms：服务器针对每个 ACK 递增 cwnd，然后发送剩余的 TCP 段。
- 264 ms：客户端收到剩余的 TCP 段，并分别发送 ACK 确认。

大家可以练习一下，如果将 cwnd 值设置为 10 个 TCP 段，那么图 2-5 所示的过程将减少一次往返，性能可以提升 22%！

通过新 TCP 连接在往返时间为 56 ms 的客户端与服务器间传输一个 20 KB 的文件需要 264 ms！ 作为对比，现在假设客户端可以重用同一个 TCP 连接（图 2-6），再发送一次相同的请求。

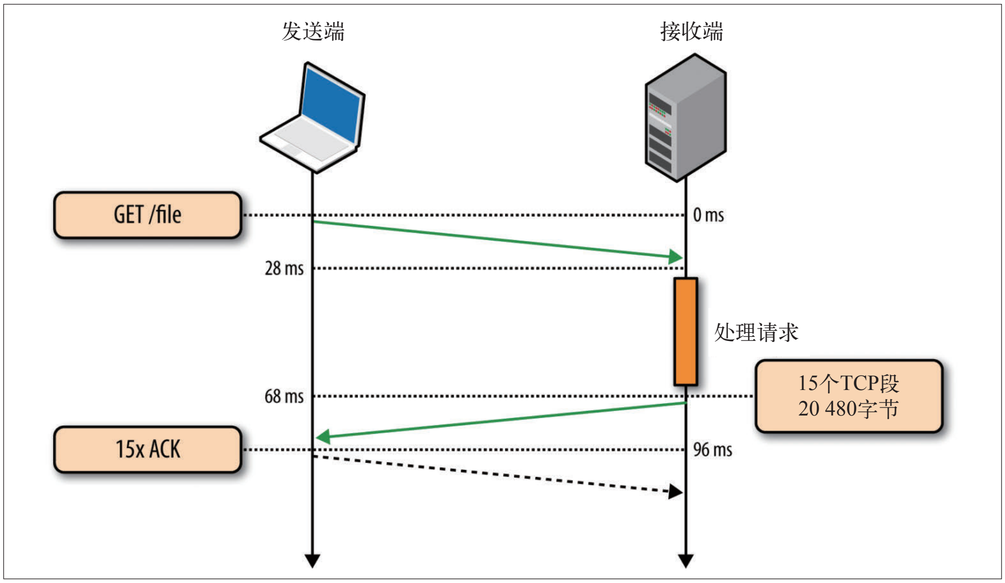


图 2-6：通过已有的 TCP 连接取得文件

- 0 ms：客户端发送 HTTP 请求。
- 28 ms：服务器收到 HTTP 请求。
- 68 ms：服务器生成 20 KB 响应，但 cwnd 已经大于发送文件所需的 15 段了，因此一次性发送所有数据段。
- 96 ms：客户端收到所有 15 个段，分别发送 ACK 确认。

同一个连接、同样的请求，但没有三次握手和慢启动，只花了 96 ms，性能提升幅度达 275%！

以上两种情况下，服务器和客户端之间的 5 Mbit/s 带宽并不影响 TCP 连接的启动阶段。此时，延迟和拥塞窗口大小才是限制因素。

事实上，如果增大往返时间，第一次和第二次请求的性能差距只会加大。大家可以练习一下，试试不同的往返时间会有什么结果。理解了 TCP 拥塞控制机制后，针对

keep-alive、流水线和多路复用的优化就简单得多了。

增大 TCP 的初始拥塞窗口

把服务器的初始 cwnd 值增大到 RFC 6928 新规定的 10 段（IW10），是提升用户体验以及所有 TCP 应用性能的最简单方式。好消息是，很多操作系统已经更新了内核，采用了增大后的值。可以留意相应的文档和发布说明。

在 Linux 上，IW10 是 2.6.39 以上版本内核的新默认值。但不要就此满足，升级到 3.2 以上版本还有其他重要更新，比如 2.2.3 节的“TCP 比例降速”。

2.2.3　拥塞预防

认识到 TCP 调节性能主要依赖丢包反馈机制非常重要。换句话说，这不是一个假设命题，而是一个具体何时发生的命题。慢启动以保守的窗口初始化连接，随后的每次往返都会成倍提高传输的数据量，直到超过接收端的流量控制窗口，即系统配置的拥塞阈值（ssthresh）窗口，或者有分组丢失为止，此时拥塞预防算法介入（图 2-3）。

拥塞预防算法把丢包作为网络拥塞的标志，即路径中某个连接或路由器已经拥堵了，以至于必须采取删包措施。因此，必须调整窗口大小，以避免造成更多的包丢失，从而保证网络畅通。

重置拥塞窗口后，拥塞预防机制按照自己的算法来增大窗口以尽量避免丢包。某个时刻，可能又会有包丢失，于是这个过程再从头开始。如果你看到过 TCP 连接的吞吐量跟踪曲线，发现该曲线呈锯齿状，那现在就该明白为什么了。这是拥塞控制和预防算法在调整拥塞窗口，进而消除网络中的丢包问题。

值得一提的是，改进拥塞控制和预防机制既是学术研究的课题，也是商业研发的方向。毕竟，有那么多不同的网络、不同的数据传输方式需要适应。今天，你的平台中可能运行着下列诸多 TCP 版本中的一个：TCP Tahoe 和 Reno（最早的实现）、TCP Vegas、TCP New Reno、TCP BIC、TCP CUBIC（Linux 的默认实现）或 Compound TCP（Windows 的默认实现），等等。然而无论你使用哪一个，拥塞控制和预防对网络性能的影响都是存在的。

TCP 比例降速

确定丢包恢复的最优方式并不容易。如果太激进，那么间歇性的丢包就会对整个连接的吞吐量造成很大影响。而如果不够快，那么还会继续造成更多分组丢失。

最初，TCP 使用 AIMD（Multiplicative Decrease and Additive Increase，倍减加增）算法，即发生丢包时，先将拥塞窗口减半，然后每次往返再缓慢地给窗口增加一个固定的值。不过，很多时候 AIMD 算法太过保守，因此又有了新的算法。

PRR（Proportional Rate Reduction，比例降速）就是 RFC 6937 规定的一个新算法，其目标就是改进丢包后的恢复速度。改进效果如何呢？根据谷歌的测量，实现新算法后，因丢包造成的平均连接延迟减少了 3%~10%。

PRR 现在是 Linux 3.2+ 内核默认的拥塞预防算法。这又是你升级服务器的一个理由！

2.3 带宽延迟积

TCP 内置的拥塞控制和预防机制对性能还有另一个重要影响：发送端和接收端理想的窗口大小，一定会因往返时间及目标传输速率而变化。

为什么？我们知道，发送端和接收端之间在途未确认的最大数据量，取决于拥塞窗口（cwnd）和接收窗口（rwnd）的最小值。接收窗口会随每次 ACK 一起发送，而拥塞窗口则由发送端根据拥塞控制和预防算法动态调整。

无论发送端发送的数据还是接收端接收的数据超过了未确认的最大数据量，都必须停下来等待另一方 ACK 确认某些分组才能继续。要等待多长时间呢？取决于往返时间！

- BDP（Bandwidth-delay product，带宽延迟积）
  数据链路的容量与其端到端延迟的乘积。这个结果就是任意时刻处于在途未确认状态的最大数据量。

发送端或接收端无论谁被迫频繁地停止等待之前分组的 ACK，都会造成数据缺口（图 2-7），从而必然限制连接的最大吞吐量。为解决这个问题，应该让窗口足够大，以保证任何一端都能在 ACK 返回前持续发送数据。只有传输不中断，才能保证最大吞吐量。而最优窗口大小取决于往返时间！无论实际或通告的带宽是多大，窗口过小都会限制连接的吞吐量。



图 2-7：拥塞窗口小导致数据缺口

那么，流量控制窗口（rwnd）和拥塞控制窗口（cwnd）的值多大合适呢？实际上，计算过程很简单。首先，假设 cwnd 和 rwnd 的最小值为 16 KB，往返时间为 100 ms：

$$16\ \text{KB} = (16 \times 1024 \times 8) = 131\,072\ \text{bit}$$
$$\frac{131\,072\ \text{bit}}{0.1\ \text{s}} = 1\,310\,720\ \text{bit/s}$$
$$1\,310\,720\ \text{bit/s} = \frac{1\,310\,720}{1\,000\,000} = 1.31\ \text{Mbit/s}$$

不管发送端和接收端的实际带宽多大，这个 TCP 连接的数据传输速率不会超过 1.31 Mbit/s！想提高吞吐量，要么增大最小窗口值，要么减少往返时间。

类似地，知道往返时间和两端的实际带宽也可以计算最优窗口大小。这一次我们假设往返时间不变（还是 100 ms），发送端的可用带宽为 10 Mbit/s，接收端则为 100 Mbit/s+。还假设两端之间没有网络拥塞，我们的目标就是充分利用客户端的 10 Mbit/s 带宽：

$$10\ \text{Mbit/s} = 10 \times 1\,000\,000 = 10\,000\,000\ \text{bit/s}$$
$$10\,000\,000\ \text{bit/s} = \frac{10\,000\,000}{8 \times 1024} = 1221\ \text{KB/s}$$
$$1221\ \text{KB/s} \times 0.1\ \text{s} = 122.1\ \text{KB}$$

窗口至少需要 122.1 KB 才能充分利用 10 Mbit/s 带宽！还记得吗，如果没有“窗口缩放（RFC 1323）”，TCP 接收窗口最大只有 64 KB。是不是该好好查查自己的客户端和服务器设置啦？

好在窗口大小的协商与调节由网络栈自动控制，应该会自动调整。但尽管如此，窗口大小有时候仍然是 TCP 性能的限制因素。如果你怎么也想不通在高速连接的客户端与服务器之间，实际传输速度只有可用带宽的几分之一，那窗口大小很可能就是罪魁祸首。要么因为某一饱和端通告的接收窗口很小，要么因为网络拥堵和丢包导致拥塞窗口重置，更可能因为流量增长过快导致对连接吞吐量施加了限制。

高速局域网中的带宽延迟积

BDP 是往返时间和目标传输速度的函数。因此，往返时间不仅在高传输延迟中是一个常见的瓶颈，就算在 LAN 中也可能是一个瓶颈！

要想在 1 ms 的往返时间内达到 1 GBit/s 的传输速度，拥塞窗口同样至少要有 122 KB。计算过程与前面类似，只不过要给目标速度多加几个零，再从往返时间中拿掉同样多个零而已。

2.4　队首阻塞

TCP 在不可靠的信道上实现了可靠的网络传输。基本的分组错误检测与纠正、按序交付、丢包重发，以及保证网络最高效率的流量控制、拥塞控制和预防机制，让 TCP 成为大多数网络应用中最常见的传输协议。

虽然 TCP 很流行，但它并不是唯一的选择，而且在某些情况下也不是最佳的选择。特别是按序交付和可靠交付有时候并不必要，反而会导致额外的延迟，对性能造成负面影响。

要理解为什么，可以想一想，每个 TCP 分组都会带着一个唯一的序列号被发出，而所有分组必须按顺序传送到接收端（图 2-8）。如果中途有一个分组没能到达接收端，那么后续分组必须保存在接收端的 TCP 缓冲区，等待丢失的分组重发并到达接收端。这一切都发生在 TCP 层，应用程序对 TCP 重发和缓冲区中排队的分组一无所知，必须等待分组全部到达才能访问数据。在此之前，应用程序只能在通过套接字读数据时感觉到延迟交付。这种效应称为 TCP 的队首（HOL，Head of Line）阻塞。

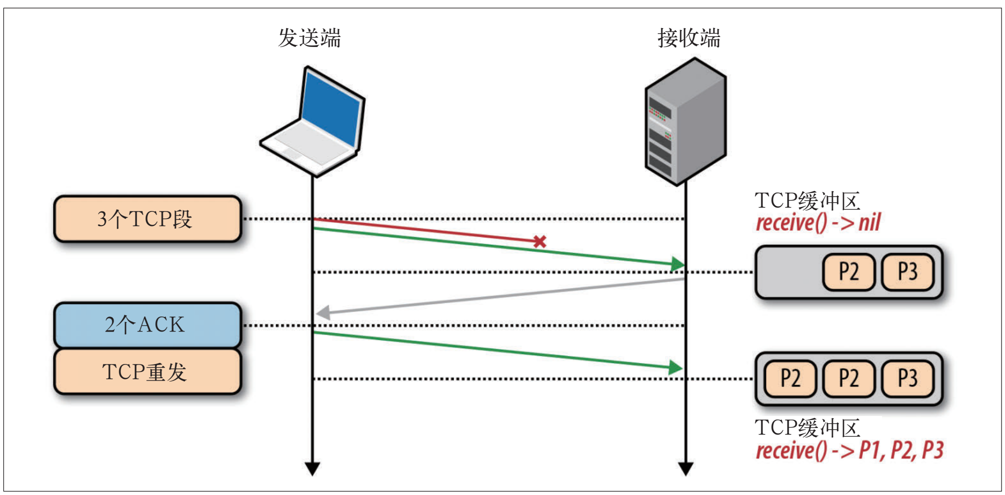


图 2-8：TCP 队首阻塞

队首阻塞造成的延迟可以让我们的应用程序不用关心分组重排和重组，从而让代码保持简洁。然而，代码简洁也要付出代价，那就是分组到达时间会存在无法预知的延迟变化。这个时间变化通常被称为抖动，也是影响应用程序性能的一个主要因素。

另外，有些应用程序可能并不需要可靠的交付或者不需要按顺序交付。比如，每个分组都是独立的消息，那么按顺序交付就没有任何必要。而且，如果每个消息都会覆盖之前的消息，那么可靠交付同样也没有必要了。可惜的是，TCP 不支持这种情

况，所有分组必须按顺序交付。

无需按序交付数据或能够处理分组丢失的应用程序，以及对延迟或抖动要求很高的应用程序，最好选择 UDP 等协议。

> **丢包就丢包**
>
> 事实上，丢包是让 TCP 达到最佳性能的关键。被删除的包恰恰是一种反馈机制，能够让接收端和发送端各自调整速度，以避免网络拥堵，同时保持延迟最短（参见 1.2 节的“本地路由器的缓冲区爆满”）。另外，有些应用程序可以容忍丢失一定数量的包，比如语音和游戏状态通信，就不需要可靠传输或按序交付。
>
> 就算有个包丢了，音频编解码器只要在音频中插入一个小小的间歇，就可以继续处理后来的包。只要间歇够小，用户就注意不到，而等待丢失的包则可能导致音频输出产生无法预料的暂停。相对来说，后者的用户体验更糟糕。
>
> 类似地，更新 3D 游戏中角色的状态也一样：收到 T 时刻的包而等待 T–1 时刻的包通常毫无必要。理想情况下，应该可以接收所有状态更新，但为避免游戏延迟，间歇性的丢包也是可以接受的。

2.5　针对TCP的优化建议

TCP 是一个自适应的、对所有网络节点一视同仁的、最大限制利用底层网络的协议。因此，优化 TCP 的最佳途径就是调整它感知当前网络状况的方式，根据它之上或之下的抽象层的类型和需求来改变它的行为。无线网络可能需要不同的拥塞算法，而某些应用程序可能需要自定义服务品质（QoS，Quality of Service）的含义，从而交付最佳的体验。

不同应用程序需求间的复杂关系，以及每个 TCP 算法中的大量因素，使得 TCP 调优成为学术和商业研究的一个“无底洞”。本章只蜻蜓点水般地介绍了影响 TCP 性能的几个典型因素，而没有探讨的选择性应答（SACK）、延迟应答、快速转发等，随便一个都能让你领略到 TCP 的复杂性（或者乐趣），感受到理解、分析和调优之难。

尽管如此，而且每个算法和反馈机制的具体细节可能会继续发展，但核心原理以及它们的影响是不变的：

- TCP 三次握手增加了整整一次往返时间；
- TCP 慢启动将被应用到每个新连接；
- TCP 流量及拥塞控制会影响所有连接的吞吐量；
- TCP 的吞吐量由当前拥塞窗口大小控制。

结果，现代高速网络中 TCP 连接的数据传输速度，往往会受到接收端和发送端之间往返时间的限制。另外，尽管带宽不断增长，但延迟依旧受限于光速，而且已经限定在了其最大值的一个很小的常数因子之内。大多数情况下，TCP 的瓶颈都是延迟，而非带宽（参见图 2-5）。

2.5.1　服务器配置调优

在着手调整 TCP 的缓冲区、超时等数十个变量之前，最好先把主机操作系统升级到最新版本。TCP 的最佳实践以及影响其性能的底层算法一直在与时俱进，而且大多数变化都只在最新内核中才有实现。一句话，让你的服务器跟上时代是优化发送端和接收端 TCP 栈的首要措施。

> 表面看来，升级服务器内核到最新版本好像是件易如反掌的事儿。但在实践中，升级经常会遭遇阻力。很多现有服务器已经针对特定的内核版本进行了调优，而系统管理员并不情愿升级。
>
> 实事求是地讲，每一次升级都有相应的风险。但为了获得最大的 TCP 性能，升级恐怕也是唯一最佳的选择。

有了最新的内核，我们推荐你遵循如下最佳实践来配置自己的服务器。

- *增大TCP的初始拥塞窗口*

  加大起始拥塞窗口可以让 TCP 在第一次往返就传输较多数据，而随后的速度提升也会很明显。对于突发性的短暂连接，这也是特别关键的一个优化。

- *慢启动重启*

  在连接空闲时禁用慢启动可以改善瞬时发送数据的长 TCP 连接的性能。

- *窗口缩放*（RFC 1323）

  启用窗口缩放可以增大最大接收窗口大小，可以让高延迟的连接达到更好吞吐量。

- *TCP快速打开*

  在某些条件下，允许在第一个 SYN 分组中发送应用程序数据。TFO（TCP Fast Open，TCP 快速打开）是一种新的优化选项，需要客户端和服务器共同支持。为此，首先要搞清楚你的应用程序是否可以利用这个特性。

以上几个设置再加上最新的内核，可以确保最佳性能：每个 TCP 连接都会具有较低的延迟和较高的吞吐量。

视应用程序的类型，可能还有必要调整服务器上的其他 TCP 设置，以便优化高速连接的速度、内存占用，或者其他类似的关键选项。不过，这些系统配置与平台、应用程序、硬件有关，超出了本书讨论范围；必要时，可以参考平台文档。但更重要的是要分清轻重缓急，着力解决真正的瓶颈，而不是眉毛胡子一把抓。

> Linux 用户可以使用 `ss` 来查看当前打开的套接字的各种统计信息。在命令行里运行 `ss --options --extended --memory --processes --info`，可以看到当前通信节点以及它们相应的连接设置。

2.5.2　应用程序行为调优

调优 TCP 性能可以让服务器和客户端之间达到最大吞吐量和最小延迟。而应用程序如何使用新的或已经建立的 TCP 连接同样也有很大的关系。

- 再快也快不过什么也不用发送，能少发就少发。
- 我们不能让数据传输得更快，但可以让它们传输的距离更短。
- 重用 TCP 连接是提升性能的关键。

当然，消除不必要的数据传输本身就是很大的优化。比如，减少下载不必要的资源，或者通过压缩算法把要发送的比特数降到最低。然后，通过在不同的地区部署服务器（比如，使用 CDN），把数据放到接近客户端的地方，可以减少网络往返的延迟，从而显著提升 TCP 性能。最后，尽可能重用已经建立的 TCP 连接，把慢启动和其他拥塞控制机制的影响降到最低。

2.5.3　性能检查清单

优化 TCP 性能的回报是丰厚的，无论什么应用，性能提升可以在与服务器的每个连接中体现出来。下面几条请大家务必记在自己的日程表里：

- 把服务器内核升级到最新版本（Linux：3.2+）；
- 确保 cwnd 大小为 10；
- 禁用空闲后的慢启动；
- 确保启动窗口缩放；
- 减少传输冗余数据；
- 压缩要传输的数据；
- 把服务器放到离用户近的地方以减少往返时间；
- 尽最大可能重用已经建立的 TCP 连接。

第3章

UDP的构成

1980 年 8 月，紧随 TCP/IP 之后，UDP（User Datagram Protocol，用户数据报协议）被 John Postel 加入了核心网络协议套件 。当时，正值 TCP 和 IP 规范分立为两个单独的 RFC。这个时间点非常重要，稍后我们会看到，UDP 的主要功能和亮点并不在于它引入了什么特性，而在于它忽略的那些特性。UDP 经常被称为无（Null）协议，RFC 768 描述了其运作机制，全文完全可以写在一张餐巾纸上。

- 数据报

  一个完整、独立的数据实体，携带着从源节点到目的地节点的足够信息，对这些节点间之前的数据交换和传输网络没有任何依赖。

数据报（datagram）和分组（packet）是两个经常被人混用的词，实际上它们还是有区别的。分组可以用来指代任何格式化的数据块，而数据报则通常只用来描述那些通过不可靠的服务传输的分组，既不保证送达，也不发送失败通知。正因为如此，很多场合下人们都把 UDP 中 User（用户）的 U，改成 Unreliable（不可靠）的 U，于是 UDP 就成了“不可靠数据报协议”（Unreliable Datagram Protocol）。这也是为什么把 UDP 分组称为数据报更为恰当的原因。

关于 UDP 的应用，最广为人知同时也是所有浏览器和因特网应用都赖以运作的，就是 DNS（Domain Name System，域名系统）。DNS 负责把对人类友好的主机名转换成 IP 地址。可是，尽管浏览器有赖于 UDP，但这个协议以前从未被看成网页和应用的关键传输机制。HTTP 并未规定要使用 TCP，但现实中所有 HTTP 实现（以及构建于其上的所有服务）都使用 TCP。

不过，这都是过去的事了。IETF 和 W3C 工作组共同制定了一套新 API——WebRTC（Web Real-Time Communication，Web 实时通信）。WebRTC 着眼于在浏览器中通过 UDP 实现原生的语音和视频实时通信，以及其他形式的 P2P（Peer-to-Peer，端到端）通信。正是因为 WebRTC 的出现，UDP 作为浏览器中重要传输机制的地位才得以突显，而且还有了浏览器 API！本书将在第 18 章再探讨 WebRTC，本章我们先来介绍一下 UDP 协议的工作原理，搞清楚为什么以及什么时候会用到它。

3.1 无协议服务

要理解为什么 UDP 被人称作“无协议”，必须从作为 TCP 和 UDP 下一层的 IP 协议说起。

IP 层的主要任务就是按照地址从源主机向目标主机发送数据报。为此，消息会被封装在一个 IP 分组内（图 3-1），其中载明了源地址和目标地址，以及其他一些路由参数。注意，数据报这个词暗示了一个重要的信息：IP 层不保证消息可靠的交付，也不发送失败通知，实际上是把底层网络的不可靠性直接暴露给了上一层。如果某个路由节点因为网络拥塞、负载过高或其他原因而删除了 IP 分组，那么在必要的情况下，IP 的上一层协议要负责检测、恢复和重发数据。

<table>
<tr><th>位</th><th colspan="2">+0..7</th><th colspan="2">+8..15</th><th colspan="2">+16..23</th><th>+24..31</th></tr>
<tr><td>0</td><td>版本</td><td>首部长度</td><td>DSCP</td><td>ECN</td><td colspan="3">总长度</td></tr>
<tr><td>32</td><td colspan="4">标识</td><td>标志</td><td colspan="2">分片偏移值</td></tr>
<tr><td>64</td><td colspan="2">存活时间</td><td colspan="2">协议</td><td colspan="3">首部校验和</td></tr>
<tr><td>96</td><td colspan="7">源IP地址</td></tr>
<tr><td>128</td><td colspan="7">目标IP地址</td></tr>
<tr><td>160</td><td colspan="7">可选项（如果有的话）</td></tr>
<tr><td>...</td><td colspan="7">净荷</td></tr>
</table>

图 3-1：IPv4 首部（20 字节）

UDP 协议会用自己的分组结构（图 3-2）封装用户消息，它只增加了 4 个字段：源端口、目标端口、分组长度和校验和。这样，当 IP 把分组送达目标主机时，该主机能够拆开 UDP 分组，根据目标端口找到目标应用程序，然后再把消息发送过去。仅此而已。

位	+0..7	+8..15	+16..23	+24..31
0	源端口		目标端口	
32	长度		校验和	
...	净荷			

图 3-2：UDP 首部（8 字节）

事实上，UDP 数据报中的源端口和校验和字段都是可选的。IP 分组的首部也有校验和，应用程序可以忽略 UDP 校验和。也就是说，所有错误检测和错误纠正工作都可以委托给上层的应用程序。说到底，UDP 仅仅是在 IP 层之上通过嵌入应用程序的源端口和目标端口，提供了一个"应用程序多路复用"机制。明白了这一点，就可以总结一下 UDP 的无服务是怎么回事了。

- *不保证消息交付*
  不确认，不重传，无超时。
- *不保证交付顺序*
  不设置包序号，不重排，不会发生队首阻塞。
- *不跟踪连接状态*
  不必建立连接或重启状态机。
- *不需要拥塞控制*
  不内置客户端或网络反馈机制。

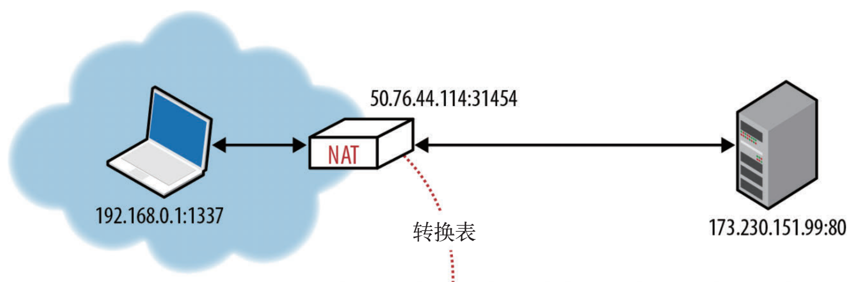
TCP 是一个面向字节流的协议，能够以多个分组形式发送应用程序消息，且对分组中的消息范围没有任何明确限制。因此，连接的两端存在一个连接状态，每个分组都有序号，丢失还要重发，并且要按顺序交付。相对来说，UDP 数据报有明确的限制：数据报必须封装在 IP 分组中，应用程序必须读取完整的消息。换句话说，数据报不能分片。

UDP 是一个简单、无状态的协议，适合作为其他上层应用协议的辅助。实际上，这个协议的所有决定都需要由上层的应用程序作出。不过，在急着去实现一个协议来扮演 TCP 的角色之前，你还应该认真想一想这里涉及的复杂细节，比如 UDP 要与很多中间设备打交道（NAT 穿透），再想一想设计网络协议的那些最佳实践。如果没有周密的设计和规划，一流的构想也可能沦为二流的 TCP 实现。TCP 中的算法和状态机已经经过了几十年的磨合与改进，而且吸收几十种并不那么容易重新实现的机制。

3.2 UDP与网络地址转换器

令人遗憾的是，IPv4 地址只有 32 位长，因而最多只能提供 42.9 亿个唯一 IP 地址。1990 年代初，互联网上的主机数量呈指数级增长，但不可能所有主机都分配一个唯一的 IP 地址。1994 年，作为解决 IPv4 地址即将耗尽的一个临时性方案，IP 网络地址转换器（NAT，Network Address Translator）规范出台了，这就是 RFC 1631。

建议的 IP 重用方案就是在网络边缘加入 NAT 设备，每个 NAT 设备负责维护一个表，表中包含本地 IP 和端口到全球唯一（外网）IP 和端口的映射（图 3-3）。这样，NAT 设备背后的 IP 地址空间就可以在各种不同的网络中得到重用，从而解决地址耗尽问题。



内网		外网	
IP	端口	IP	端口
192.168.0.1	1337	50.76.44.114	31454
...	...	...	...

图 3-3：IP 网络地址转换器

然而，这个临时性的方案居然就那么一直沿用了下来（这种现象倒也不鲜见）。新增的 NAT 设备不仅立杆见影地解决了地址耗尽问题，而且迅速成为很多公司及家庭代理和路由器、安全装置、防火墙，以及其他许多硬件和软件设备中的内置组件。NAT 不再是个临时性方案，它已经成了因特网基础设施的一个组成部分。

保留的私有网络地址范围

作为监管全球 IP 地址分配的机构，IANA（Internet Assigned Numbers Authority，因特网号码分配机构）为私有网络保留了三段 IP 地址，这些 IP 地址经常可以在 NAT 设备后面的内网中看到。

表3-1：保留的IP地址范围

IP地址范围	地址数量
10.0.0.0~10.255.255.255	16 777 216
172.16.0.0~172.31.255.255	1 048 576
192.168.0.0~192.168.255.255	65 536

其中一段（或所有三段）IP 地址是不是很眼熟？在你的局域网中，路由器给你的计算机分配的 IP 地址很可能位于其中一段。这个地址就是你在内网中的私有地址。在需要与外网通信时，NAT 设备会将它们转换成外网地址。

为防止路由错误和引起不必要的麻烦，不允许给外网计算机分配这些保留的私有 IP 地址。

3.2.1 连接状态超时

NAT 转换的问题（至少对于 UDP 而言）在于必须维护一份精确的路由表才能保证数据转发。NAT 设备依赖连接状态，而 UDP 没有状态。这种根本上的错配是很多 UDP 数据报传输问题的总根源。况且，客户端前面有很多个 NAT 设备的情况也不鲜见，问题由此进一步恶化了。

每个 TCP 连接都有一个设计周密的协议状态机，从握手开始，然后传输应用数据，最后通过明确的信号确认关闭连接。在这种设计下，路由设备可以监控连接状态，根据情况创建或删除路由表中的条目。而 UDP 呢，没有握手，没有连接终止，实际根本没有可监控的连接状态机。

发送出站 UDP 不费事，但路由响应却需要转换表中有一个条目能告诉我们本地目标主机的 IP 和端口。因此，转换器必须保存每个 UDP 流的状态，而 UDP 自身却没有状态。

更糟糕的是，NAT 设备还被赋予了删除转换记录的责任，但由于 UDP 没有连接终止确认环节，任何一端随时都可以停止传输数据报，而不必发送通告。为解决这个问题，UDP 路由记录会定时过期。定时多长？没有规定，完全取决于转换器的制造商、型号、版本和配置。因此，对于较长时间的 UDP 通信，有一个事实上的最佳做法，即引入一个双向 keep-alive 分组，周期性地重置传输路径上所有 NAT 设备中转换记录的计时器。

TCP 超时和 NAT

从技术角度讲，NAT 设备不需要额外的 TCP 超时机制。TCP 协议就遵循一个设计严密的握手与终止过程，通过这个过程就可以确定何时需要添加或删除转换记录。遗憾的是，实际应用中的 NAT 设备给 TCP 和 UDP 会话应用了类似的超时逻辑。

这样就导致 TCP 连接有时候也需要双向 keep-alive 分组。如果你的 TCP 连接突然断开，那很有可能就是中间 NAT 超时造成的。

3.2.2 NAT穿透

不可预测的连接状态处理是 NAT 设备带来的一个严重问题，但更为严重的则是很多应用程序根本就不能建立 UDP 连接。尤其是 P2P 应用程序，涉及 VoIP、游戏和文件共享等，它们客户端与服务器经常需要角色互换，以实现端到端的双向通信。

NAT 带来的第一个问题，就是内部客户端不知道外网 IP 地址，只知道内网 IP 地址。NAT 负责重写每个 UDP 分组中的源端口、地址，以及 IP 分组中的源 IP 地址。如果客户端在应用数据中以其内网 IP 地址与外网主机通信，连接必然失败。所谓的“透明”转换因此也就成了一句空话，如果应用程序想与私有网络外部的主机通信，那么它首先 必须知道自己的外网 IP 地址。

然而，知道外网 IP 地址还不是实现 UDP 传输的充分条件。任何到达 NAT 设备外网 IP 的分组还必须有一个目标端口，而且 NAT 转换表中也要有一个条目可以将其转换为内部主机的 IP 地址和端口号。如果没有这个条目（通常是从外网传数据进来），那到达的分组就会被删除（图 3-4）。此时的 NAT 设备就像一个分组过滤器，除非用户通过端口转发（映射）或类似机制配置过，否则它无法确定将 . 分组发送给哪台内部主机。

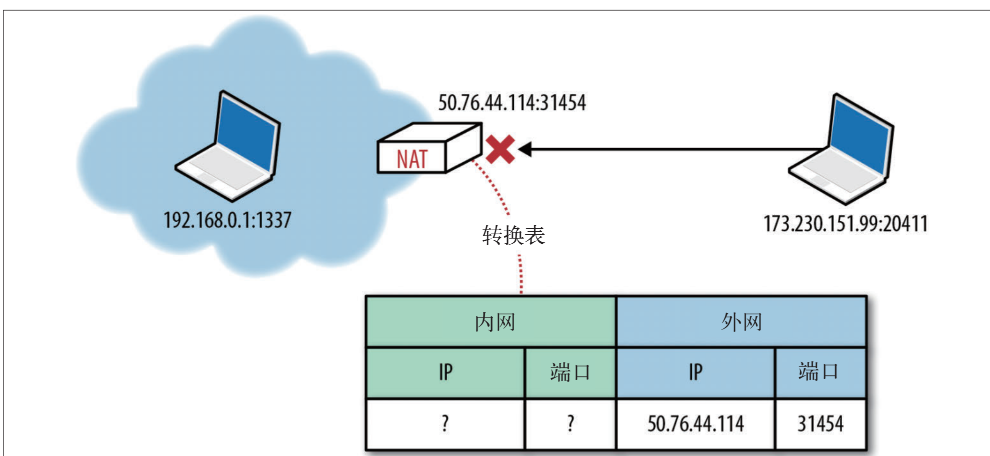


图 3-4：由于没有映射规则，入站分组直接被删除

需要注意的是，上述行为对客户端应用程序不是问题。客户端应用程序基于内部网络实现交互，会在交互期间建立必要的转换记录。不过，如果隔着 NAT 设备，那客户端（作为服务器）处理来自 P2P 应用程序（VoIP、游戏、文件共享）的入站连接时，就必须面对 NAT 穿透问题。

为解决 UDP 与 NAT 的这种不搭配，人们发明了很多穿透技术（TURN、STUN、ICE），用于在 UDP 主机之间建立端到端的连接。

3.2.3 STUN、TURN与ICE

STUN（Session Traversal Utilities for NAT）是一个协议（RFC 5389），可以让应用程序发现网络中的地址转换器，发现之后进一步取得为当前连接分配的外网 IP 地址和端口（图 3-5）。为此，这个协议需要一个已知的第三方 STUN 服务器支持，该服务器必须架设在公网上。

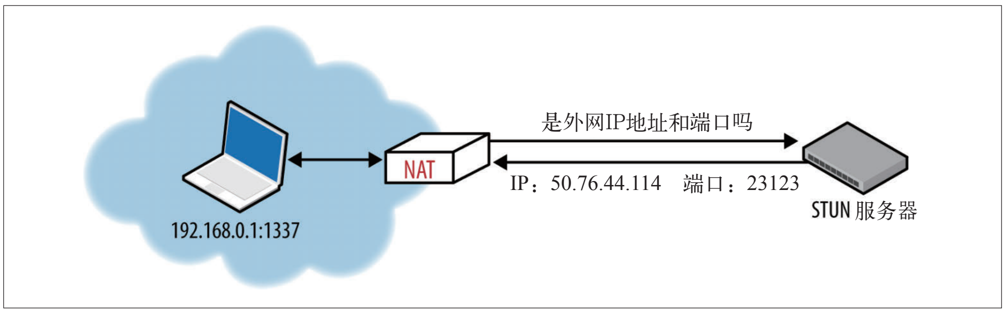


图 3-5：STUN 查询外网 IP 地址和端口

假设 STUN 服务器的 IP 地址已知（通过 DNS 查找或手工指定），应用程序首先向 STUN 服务器发送一个绑定请求。然后，STUN 服务器返回一个响应，其中包含在外网中代表客户端的 IP 地址和端口号。这种简单的方式解决了前面讨论的一些问题：

- 应用程序可以获得外网 IP 和端口，并利用这些信息与对端通信；
- 发送到 STUN 服务器的出站绑定请求将在通信要经过的 NAT 中建立路由条目，使得到达该 IP 和端口的入站分组可以找到内网中的应用程序；
- STUN 协议定义了一个简单 keep-alive 探测机制，可以保证 NAT 路由条目不超时。

有了这个机制，两台主机端需要通过 UDP 通信时，它们首先都会向各自的 STUN 服务器发送绑定请求，然后分别使用响应中的外网 IP 地址和端口号交换数据。

但在实际应用中，STUN 并不能适应所有类型的 NAT 和网络配置。不仅如此，某些情况下 UDP 还会被防火墙或其他网络设备完全屏蔽。这种情况在很多企业网很常

见。为解决这个问题，在 STUN 失败的情况下，我们还可以使用 TURN（Traversal Using Relays around NAT）协议（RFC 5766）作为后备。TURN 可以在最坏的情况下跳过 UDP 而切换到 TCP。

TURN 中的关键词当然是中继（relay）。这个协议依赖于外网中继设备（图 3-6）在两端间传递数据。

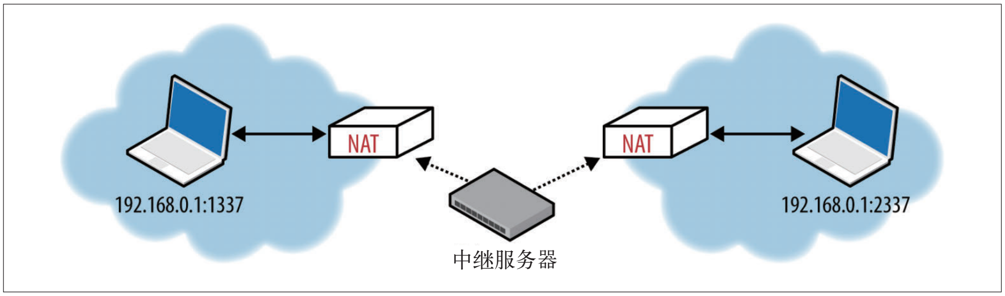


图 3-6：TURN 中继服务器

- 两端都要向同一台 TURN 服务器发送分配请求来建立连接，然后再进行权限协商。
- 协商完毕，两端都把数据发送到 TURN 服务器，再由 TURN 服务器转发，从而实现通信。

很明显，这就不再是端对端的数据交换了！ TURN 是在任何网络中为两端提供连接的最可靠方式，但运维 TURN 服务器的投入也很大。至少，为满足传输数据的需要，中继设备的容量必须足够大。因此，最好在其他直连手段都失败的情况下，再使用 TURN。

现实中的 STUN 和 TURN

谷歌的 libjingle 是一个用 C++ 开发的用于构建端到端应用程序的开源库，负责在后台实现 STUN、TURN 和 ICE 协商。谷歌聊天软件 Google Talk 使用的就是这个库，其文档也为我们考量现实中的 STUN 与 TURN 性能提供了有价值的参考：

- 92% 的时间可以直接连接（STUN）；
- 8% 的时间要使用中继器（TURN）。

由于 NAT 设备遍地都是，相当一部分用户不能通过 STUN 直接建立 P2P 连接。若要提供可靠的 UDP 服务，现实中必须同时支持 STUN 和 TURN。

构建高效的 NAT 穿透方案可不容易。好在，我们还有 ICE（Interactive Connectivity Establishment）协议（RFC 5245）。ICE 规定了一套方法，致力于在通信各端之间建立一

条最有效的通道（图 3-7）：能直连就直连，必要时 STUN 协商，再不行使用 TURN。

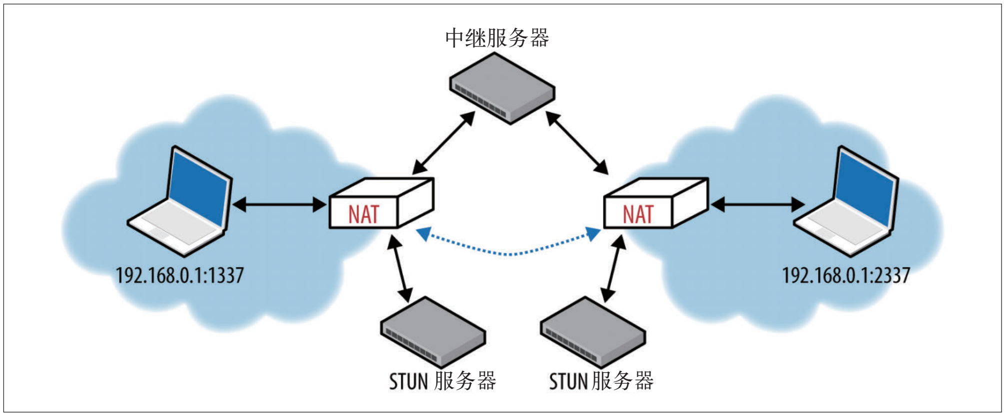


图 3-7：ICE 先后尝试直连、STUN 和 TURN

实际开发中，如果你想构建基于 UDP 的 P2P 应用程序，绝对应该选择现有的平台 API，或者实现了 ICE、STUN 和 TURN 的第三方库。好了，了解了这些协议的用途之后，接下来自然就要考虑安装和配置了！

3.3　针对UDP的优化建议

UDP 是一个简单常用的协议，经常用于引导其他传输协议。事实上，UDP 的特色在于它所省略的那些功能：连接状态、握手、重发、重组、重排、拥塞控制、拥塞预防、流量控制，甚至可选的错误检测，统统没有。这个面向消息的最简单的传输层在提供灵活性的同时，也给实现者带来了麻烦。你的应用程序很可能需要从头实现上述几个或者大部分功能，而且每项功能都必须保证与网络中的其他主机和协议和谐共存。

与内置流量和拥塞控制以及拥塞预防的 TCP 不同，UDP 应用程序必须自己实现这些机制。拥塞处理做得不到位的 UDP 应用程序很容易堵塞网络，造成网络性能下降，严重时还会导致网络拥塞崩溃。如果你想在自己的应用程序中使用 UDP，务必要认真研究和学习当下的最佳实践和建议。RFC 5405 就是这么一份文档，它对设计单播 UDP 应用程序给出了很多设计建议，简述如下：

- 应用程序必须容忍各种因特网路径条件；
- 应用程序应该控制传输速度；
- 应用程序应该对所有流量进行拥塞控制；
- 应用程序应该使用与 TCP 相近的带宽；

- 应用程序应该准备基于丢包的重发计数器；
- 应用程序应该不发送大于路径 MTU 的数据报；
- 应用程序应该处理数据报丢失、重复和重排；
- 应用程序应该足够稳定以支持 2 分钟以上的交付延迟；
- 应用程序应该支持 IPv4 UDP 校验和，必须支持 IPv6 校验和；
- 应用程序可以在需要时使用 keep-alive（最小间隔 15 秒）。

设计新传输协议必须经过周密的考虑、规划和研究，否则就是不负责任。要尽可能利用已有的库或框架，这个库或框架应该考虑了 NAT 穿透，而且能够与其他并发的网络流量和谐共存。

我很高兴告诉大家：WebRTC 就是符合这些要求的框架！

第 4 章

传输层安全（TLS）

SSL（Secure Sockets Layer，安全套接字层）协议最初是网景公司为了保障网上交易安全而开发的，该协议通过加密来保护客户个人资料，通过认证和完整性检查来确保交易安全。为达到这个目标，SSL 协议在直接位于 TCP 上一层的应用层被实现（图 4-1）。SSL 不会影响上层协议（如 HTTP、电子邮件、即时通讯），但能够保证上层协议的网络通信安全。

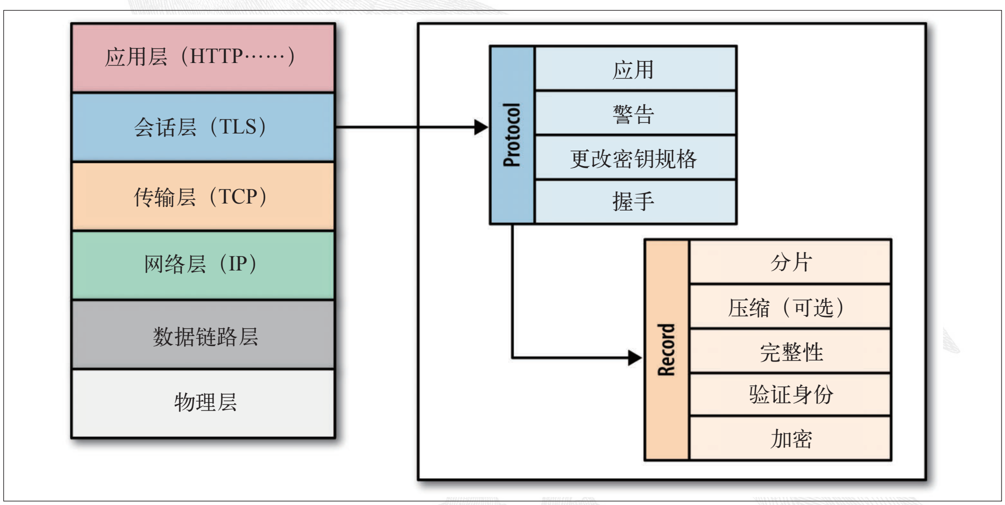


图 4-1：传输层安全（TLS）

在正确使用 SSL 的情况下，第三方监听者只能推断出连接的端点、加密类型，以及发送数据的频率和大致数量，不能实际读取或修改任何数据。

IETF 后来在标准化 SSL 协议时，将其改名为 Transport Layer Security（TLS，传输层安全）。很多人会混用 TLS 和 SSL，但严格来讲它们并不相同，因为它们指代的协议版本不同。

SSL 2.0 是该协议第一个公开发布的版本，但由于存在很多安全缺陷很快就被 SSL 3.0 取代。鉴于 SSL 协议是网景公司专有的，IETF 成立了一个小组负责标准化该协议，后来就有了 RFC 2246，即 TLS 1.0，也就是 SSL 3.0 的升级版。

> 本协议与 SSL 3.0 的区别并不特别明显，但这些区别会严重妨碍 TLS 1.0 与 SSL 3.0 之间的互操作性。
>
> ——TLS 协议（RFC 2246）

TLS 1.0 自 1999 年 1 月发布后，为解决发现的安全缺陷同时扩展协议的功能，IETF 工作组先后又发布过两个新版本：2006 年 4 月发布了 TLS 1.1，2008 年 8 月发布了 TLS 1.2。从内部来看，SSL 3.0 实现与后续所有 TLS 版本很相似，很多客户端到今天还在支持 SSL 3.0 和 TLS 1.0。当然，要保护用户免受攻击，最好还是升级到最新版本。

TLS 设计的初衷是在可靠的传输协议（如 TCP）之上运行。可是，有实现把它放到了数据报协议（如 UDP）之上。RFC 6347，即 DTLS（Datagram Transport Layer Security，数据报传输层安全）就旨在以 TLS 协议为基础，同时兼顾数据报交付模式并提供类似的安全保障。

4.1　加密、身份验证与完整性

TLS 协议的目标是为在它之上运行的应用提供三个基本服务：加密、身份验证和数据完整性。从技术角度讲，并不是所有情况下都要同时使用这三个服务。比如，可以接受证书但不验证其真实性，而前提是你非常清楚这样做有什么安全风险且有防范措施。实践中，安全的 Web 应用都会利用这三个服务。

- 加密
  混淆数据的机制
- 身份验证
  验证身份标识有效性的机制
- 完整性
  检测消息是否被篡改或伪造的机制

为了建立加密的安全数据通道，连接双方必须就加密数据的密钥套件和密钥协商一致 。TLS 协议规定了一套严密的握手程序用于交换这些信息，相关内容将在 4.2 节“TLS 握手”中介绍。握手机制中设计最巧妙的地方，就是其使用的公钥密码系统（也称“非对称密钥加密”），这套系统可以让通信双方不必事先“认识”即可商定共享的安全密钥，而且协商过程还是通过非加密通道完成的。

握手过程中，TLS 协议还允许通信两端互相验明正身。在浏览器中，验证机制允许客户端验证服务器就是它想联系的那个（比如，银行），而不是通过名字或 IP 地址伪装的目标。这个验证首先需要建立“认证机构信任链”（Chain of Trust and Certificate Authorities）。此外，服务器也可以选择验证客户端的身份。比如，公司的代理服务器可以验证所有雇员，每位雇员都应该有公司签发的独一无二的认证证书。

除了密钥协商和身份验证，TLS 协议还提供了自己的消息分帧机制，使用 MAC（Message Authentication Code，消息认证码）签署每一条消息。MAC 算法是一个单向加密的散列函数（本质上是一个校验和），密钥由连接双方协商确定。只要发送 TLS 记录，就会生成一个 MAC 值并附加到该消息中。接收端通过计算和验证这个 MAC 值来判断消息的完整性和可靠性。

上述三种机制为 Web 通信构建了一个安全的环境。所有现代 Web 浏览器都支持多种加密套件，能够验证客户端和服务器，并能对每条记录进行消息完整性检查。

Web 代理、中间设备、TLS 与新协议

HTTP 良好的扩展能力和获得的巨大成功，使得 Web 上出现了大量代理和中间设备：缓存服务器、安全网关、Web 加速器、内容过滤器，等等。有时候，我们知道这些设备的存在（显式代理），而有时候，这些设备对终端用户则完全不可见。

然而，这些服务器的存在及成功也给那些试图脱离 HTTP 协议的人带了一些不便。比如，有的代理服务器只会简单地转发自己无法解释的 HTTP 扩展或其他在线格式（wire format），而有的则不管是否必要都会对所有数据执行自己设定的逻辑，还有一些安全设备可能会把本来正常的数据误判成恶意通信。

换句话说，现实当中如果想脱离 HTTP 和 80 端口的语义行事，经常会遭遇各种部署上的麻烦。比如，某些客户端表现正常，另一些可能就会异常，甚至在某个网段表现正常的客户端到了另一个网段又会变得异常。

为解决这些问题，出现了一些新协议和对 HTTP 的扩展，比如 WebSocket、SPDY 等。这些新协议一般要依赖于建立 HTTPS 信道，以绕过中间代理，从而实现可靠的部署，因为加密的传输信道会对所有中间设备都混淆数据。这样虽然解决了中间设备的问题，但却导致通信两端不能再利用这些中间设备，从而与这些设备提供的身份验证、缓存、安全扫描等功能失之交臂。

> 你或许一直想不通为什么大多数 WebSocket 手册都告诉你要使用 HTTPS 向移动客户端发送数据，现在你应该明白了。随着时间推移，网络上的中间设备通过升级也开始能识别新协议，基于 HTTPS 部署的要求也将逐渐弱化。到时候，除非你的会话真的需要 TLS 提供的加密、身份验证和完整性检查功能，否则完全可以不用 HTTPS！

4.2　TLS握手

客户端与服务器在通过 TLS 交换数据之前，必须协商建立加密信道。协商内容包括 TLS 版本、加密套件，必要时还会验证证书。然而，协商过程的每一步都需要一个分组在客户端和服务器之间往返一次（图 4-2），因而所有 TLS 连接启动时都要经历一定的延迟。

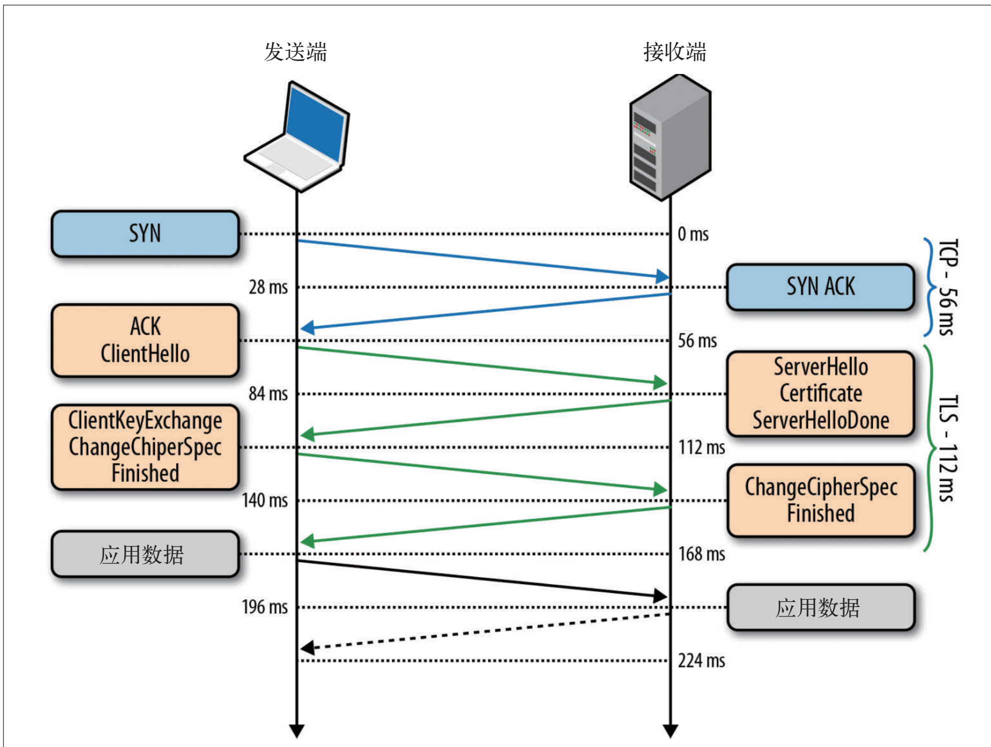


图 4-2：TLS 握手协议

> 图 4-2 假设“光通过光纤”的单程时间都是 28 ms，也就是前面 TCP 连接中从纽约到伦敦之间的时间，另见表 1-1。

- 0 ms：TLS 在可靠的传输层（TCP）之上运行，这意味着首先必须完成 TCP 的“三次握手”，即一次完整的往返。
- 56 ms：TCP 连接建立之后，客户端再以纯文本形式发送一些规格说明，比如它所运行的 TLS 协议的版本、它所支持的加密套件列表，以及它支持或希望使用的另外一些 TLS 选项。
- 84 ms：然后，服务器取得 TLS 协议版本以备将来通信使用，从客户端提供的加密套件列表中选择一个，再附上自己的证书，将响应发送回客户端。作为可选项，服务器也可以发送一个请求，要求客户端提供证书以及其他 TLS 扩展参数。
- 112 ms：假设两端经过协商确定了共同的版本和加密套件，客户端也高高兴兴地把自己的证书提供给了服务器。然后，客户端会生成一个新的对称密钥，用服务器的公钥来加密，加密后发送给服务器，告诉服务器可以开始加密通信了。到目前为止，除了用服务器公钥加密的新对称密钥之外，所有数据都以明文形式发送。
- 140 ms：最后，服务器解密出客户端发来的对称密钥，通过验证消息的 MAC 检测消息完整性，再返回给客户端一个加密的“Finished”消息。
- 168 ms：客户端用它之前生成的对称密钥解密这条消息，验证 MAC，如果一切顺利，则建立信道并开始发送应用数据。

> 新 TLS 连接要完成一次“完整的握手”需要两次网络往返。另外，还可以使用“简短握手”，只需一次往返，详细信息请参见 4.3 节“TLS 会话恢复”。

协商建立 TLS 安全信道是一个复杂的过程，很容易出错。好在服务器和浏览器会替我们做好这些工作，我们要做的就是提供和配置证书！

总之，尽管我们的 Web 应用不一定参与上述过程，但最重要的是知道每一个 TLS 连接在 TCP 握手基础上最多还需要两次额外的往返。这些都会增加实际交换数据之前的等待时间！如果考虑不周，通过 TLS 交付数据很可能会引入几百甚至几千 ms 的网络延迟。

公钥与对称密钥加密的性能

公钥加密系统（http://en.wikipedia.org/wiki/Public-key_cryptography）只在建立 TLS 信道的会话中使用。在此期间，服务器向客户端提供它的公钥，客户端生成对称密钥并使用服务器的公钥对其加密，然后再将加密的对称密钥返回服务器。服务器继而用自己的私钥解密出客户端发来的对称密钥。

接下来，客户端与服务器间的通信就全都使用客户端生成的共享密钥加密，这就是对称密钥加密。之所以这样设计，很大程度上是出于性能考虑，因为公钥加密需要很大的计算量。为了演示两者的差别，假如你的电脑上安装了 OpenSSL，可以试试以下两条命令：

- `$> openssl speed rsa`
- `$> openssl speed aes`

需要注意，这两条命令涉及的单位不能直接相比：RSA 测试会提供一个摘要表格，列出不同密钥大小每秒的操作数，而 AES 性能通过每秒的字节数来度量。无论如何，都不难看出 RSA 操作（完整的 TLS 握手）的数量，在推荐的 1024 或 2048 位的密钥长度下都很可能成为瓶颈。

实际的数值可能会因硬件、核心数量、TLS 版本、服务器配置、典型的服务负载，以及其他因素不同而有很大差异。请大家不要轻信一些广告宣传或过时的测试报告，最好还是在自己的硬件上运行上述测试。

4.2.1　应用层协议协商（ALPN）

理论上，网络上的任意两端都可以使用自定义的协议进行通信。为此，需要提前确定使用什么协议、指定端口号（HTTP 是 80，TLS 是 443），并配置所有客户端和服务器使用它们。然而在实践中，这种方法效率很低且很难做到。因为每个端口都必须得到认可，而防火墙及其他中间设备通常只允许在 80 和 443 端口上通信。

于是，为了简化自定义协议的部署，我们往往必须重用 80 或 443 端口，再通过额外的机制协商确定协议。80 端口是为 HTTP 保留的，而 HTTP 规范还专门为协商协议规定了一个 `Upgrade` 首部。可是，使用 `Upgrade` 需要一次额外的往返时间，且由于很多中间设备的存在，协商结果也不可靠。要了解更多信息，请参考 4.1 节中的“Web 代理、中间设备、TLS 与新协议”。

> 12.3.9 节“有效的 HTTP 2.0 升级与发现”中有一个使用 HTTP `Upgrade` 的实际例子。

解决办法不难想象，那就是使用 443 端口，这是给（运行于 TLS 之上的）安全 HTTPS 会话保留的。由于端到端的加密信道对中间设备模糊了数据，因此这种方式就成为了一种部署任意新应用协议的可靠而快捷的方式。不过，虽然使用 TLS 保障了可靠性，但我们还需要一种机制来协商协议。

作为 HTTPS 会话，当然可以利用 HTTP 的 `Upgrade` 机制来协商，但这会导致一次额外的往返，造成延迟。那在 TLS 握手的同时协商确定协议可行吗？

顾名思义，应用层协议协商（ALPN，Application Layer Protocol Negotiation）作为 TLS 扩展，让我们能在 TLS 握手的同时协商应用协议（图 4-2），从而省掉了 HTTP 的 `Upgrade` 机制所需的额外往返时间。具体来说，整个过程分如下几步：

- 客户端在 `ClientHello` 消息中追加一个新的 `ProtocolNameList` 字段，包含自己支持的应用协议；
- 服务器检查 `ProtocolNameList` 字段，并在 `ServerHello` 消息中以 `ProtocolName` 字段返回选中的协议。

服务器可以从中选择一个协议名，否则如果不支持其中的任何协议，则断开连接。只要 TLS 握手完成、建立了加密信道并就应用协议达成一致，客户端与服务器就可以立即通信。

> ALPN 抛开了 HTTP 的 `Upgrade` 机制，省却了一次往返的延迟。不过，TLS 握手还是必需的。事实上，ALPN 协商不会比基于非加密信道的 HTTP `Upgrade` 更快，它只能保证通过 TLS 进行协议协商不会更慢。

NPN 与 ALPN 的渊源

NPN（Next Protocol Negotiation，下一代协议协商）是谷歌在 SPDY 协议中开发的一个 TLS 扩展，目的是通过在 TLS 握手期间协商应用协议来提高效率。听着耳熟吗？最终结果与 ALPN 功能等价。

ALPN 是 IETF 在 NPN 基础上修订并批准的版本。在 NPN 中，服务器广播自己支持的协议，客户端选择和确认协议。而在 ALPN 中，交换次序颠倒过来了，客户端先声明自己支持的协议，服务器选择并确认协议。这样修改的目的是为了让 ALPN 与其他协议协商标准保持一致。

换句话说，ALPN 是 NPN 的继任者，而 NPN 已经废弃。原来配置为使用 NPN 的客户端和服务器，都需要重新配置升级到 ALPN。

4.2.2　服务器名称指示（SNI）

网络上可以建立 TCP 连接的任意两端都可以建立加密 TLS 信道，客户端只需知道另一端的 IP 地址即可建立连接并进行 TLS 握手。然而，如果服务器想在一个 IP 地址为多个站点提供服务，而每个站点都拥有自己的 TLS 证书，那怎么办？这问题看似有点怪，但其实一点都不怪。

为了解决这个问题，SNI（Server Name Indication，服务器名称指示）扩展被引入 TLS 协议，该扩展允许客户端在握手之初就指明要连接的主机名。Web 服务器可以

检查 SNI 主机名，选择适当的证书，继续完成握手。

> **TLS、HTTP 及专用 IP**
>
> TLS+SNI 机制与 HTTP 中发送 `Host` 首部是相同的，只不过后者是客户端要在请求中包含站点的主机名。总之，都是相同的 IP 地址服务于不同的域名，而区分不同域名的手段就是 SNI 或 `Host`。
>
> 遗憾的是，很多旧版本的客户端（Windows XP 上的大多数 IE、Android 2.2 等平台上的浏览器）都不支持 SNI。如果你想与这些旧版本的客户端进行 TLS 通信，就得为每个主机准备一个专用 IP 地址。

4.3 TLS会话恢复

完整 TLS 握手会带来额外的延迟和计算量，从而给所有依赖安全通信的应用造成严重的性能损失。为了挽回某些损失，TLS 提供了恢复功能，即在多个连接间共享协商后的安全密钥。

4.3.1 会话标识符

最早的“会话标识符”（Session Identifier，RFC 5246）机制是在 SSL 2.0 中引入的，支持服务器创建 32 字节的会话标识符，并在上一节我们讨论的完整的 TLS 协商期间作为其“ServerHello”消息的一部分发送。

在内部，服务器会为每个客户端保存一个会话 ID 和协商后的会话参数。相应地，客户端也可以保存会话 ID 信息，并将该 ID 包含在后续会话的“ClientHello”消息中，从而告诉服务器自己还记着上次握手协商后的加密套件和密钥呢，这些都可以重用。假设客户端和服务器都可以在自己的缓存中找到共享的会话 ID 参数，那么就可以进行简短握手（图 4-3）。否则，就要重新启动一次全新的会话协商，生成新的会话 ID。

借助会话标识符可以节省一次往返，还可以省掉用于协商共享加密密钥的公钥加密计算。由于重用了之前协商过的会话数据，就可以迅速建立一个加密连接，而且同样安全。

> 实际应用中，大多数 Web 应用会尝试与同一个主机建立多个连接，以便并行取得资源。在这种情况下，会话恢复就成为减少延迟及两端计算量的必备优化手段。
>
> 大多数现代浏览器在打开到相同服务器的新连接之前，都会有意等待第一个 TLS 连接完成。这样，后续的 TLS 连接就可以重用第一个 SSL 会话，从而避免重新握手造成的损失。

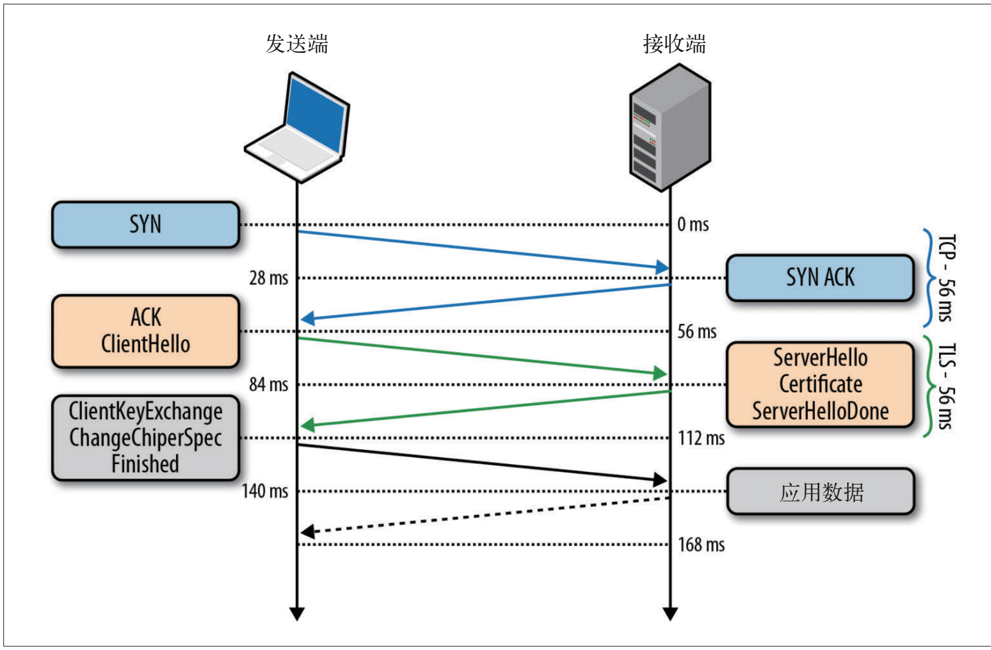


图 4-3：简短 TLS 握手协议

由于服务器必须为每个客户端都创建和维护一段会话缓存，“会话标识符”机制在实践中也会遇到问题，特别是对那些每天都要“接待”几万甚至几百万独立连接的服务器来说，问题就更多了。由于每个打开的 TLS 连接都要占用内存，因此需要一套会话 ID 缓存和清除策略，对于拥有很多服务器而且为获得最佳性能必须使用共享 TLS 会话缓存的热门站点而言，部署这些策略绝非易事。

当然，这些问题并非无解，今天的很多高流量站点都在成功地使用会话标识符。只不过对于多服务器部署来说，为确保会话缓存的良性循环，必须对会话标识符进行周密的规划和系统的设计。

4.3.2　会话记录单

为了解决上述服务器端部署 TLS 会话缓存的问题，“会话记录单”（Session Ticket，RFC 5077）机制出台了，该机制不用服务器保存每个客户端的会话状态。相反，如果客户端表明其支持会话记录单，则服务器可以在完整 TLS 握手的最后一次交换中添加一条“新会话记录单”（New Session Ticket）记录，包含只有服务器知道的安全密钥加密过的所有会话数据。

然后，客户端将这个会话记录单保存起来，在后续会话的 `ClientHello` 消息中，可

以将其包含在 `SessionTicket` 扩展中。这样，所有会话数据只保存在客户端，而由于数据被加密过，且密钥只有服务器知道，因此仍然是安全的。

我们这里所说的会话标识符和会话记录单机制，也经常被人称为“会话缓存”或“无状态恢复”机制。无状态恢复机制的优点主要是消除了服务器端的缓存负担，通过要求客户端在与服务器建立新连接时提供会话记录单简化了部署（除非记录单过期）。

> 实践中，在一组负载均衡服务器上部署会话记录单仍然要求周密规划和系统设计。比如，所有服务器一开始都拥有相同的会话密钥，然后再通过另外的机制定期在所有服务器端轮换共享的密钥。

4.4　信任链与证书颁发机构

身份验证是建立每个 TLS 连接必不可少的部分。毕竟，加密信道两端可以是任何机器，包括攻击者的机器。为此，必须确保我们与之交谈的计算机是可信任的，否则之前的工作都是徒劳。为理解如何验证通信两端的身份，下面我们以张三和李四之间的验证为例简单说明一下：

- 张三和李四分别生成自己的公钥和私钥；
- 张三和李四分别隐藏自己的私钥；
- 张三向李四公开自己的公钥，李四也向张三公开自己的公钥；
- 张三向李四发送一条新消息，并用自己的私钥签名；
- 李四使用张三的公钥验证收到的消息签名。

信任是上述交流的关键。公钥加密可以让我们使用发送端的公钥验证消息是否使用了正确的私钥签名，但认可发送端仍然是基于信任。在上述交流中，张三和李四可以当面交换自己的公钥，因为他们互相认识，能够保证不被别人冒名顶替。可以说，他们已经通过之前安全（物理）的握手确认了对方。

接下来，张三收到王五发来的一条消息。张三从未见过王五，但王五自称是李四的朋友。事实上，为了证明自己是李四的朋友，王五还请李四用李四的私钥签署了自己的公钥，并在消息中附上了签名（图 4-4）。此时，张三首先检查王五公钥中李四的签名。他知道李四的公钥，因而可以验证李四确实签署了王五的公钥。由于他信任李四对王五的签名，所以就接受了王五的消息，并对消息进行完整性检查，以确保消息确实来自王五。

刚才这个过程建立了一个信任链：张三信任李四，李四信任王五，通过信任的传递，张三信任王五。只要这条链上的人不会被冒名顶替，我们就可以继续扩展这个信任网络。

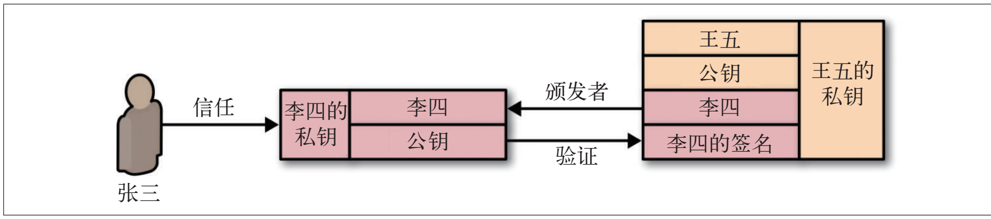


图 4-4：张三、李四和王五的信任链

Web 以及浏览器中的身份验证与上述过程相同，这就意味着此时此刻你应该问自己：我的浏览器信任谁？我在使用浏览器的时候信任谁？这个问题至少有三个答案。

- 手工指定证书

  所有浏览器和操作系统都提供了一种手工导入信任证书的机制。至于如何获得证书和验证完整性则完全由你自己来定。

- 证书颁发机构

  CA（Certificate Authority，证书颁发机构）是被证书接受者（拥有者）和依赖证书的一方共同信任的第三方。

- 浏览器和操作系统

  每个操作系统和大多数浏览器都会内置一个知名证书颁发机构的名单。因此，你也会信任操作系统及浏览器提供商提供和维护的可信任机构。

实践中，保存并手工验证每个网站的密钥是不可行的（当然，如果你愿意，也可以）。现实中最常见的方案就是让证书颁发机构替我们做这件事（图 4-5）：浏览器指定可信任的证书颁发机构（根 CA），然后验证他们签署的每个站点的责任就转移到了他们头上，他们会审计和验证这些站点的证书没有被滥用或冒充。持有 CA 证书的站点的安全性如果遭到破坏，那撤销该证书也是证书颁发机构的责任。

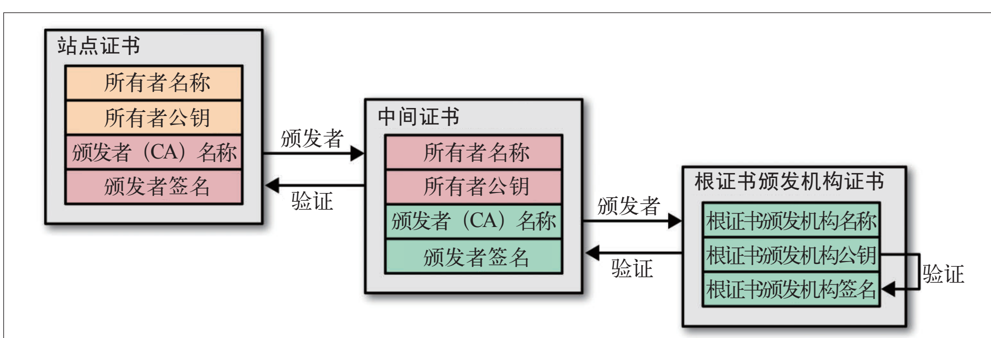


图 4-5：证书颁发机构签署数字证书

所有浏览器都允许用户检视自己安全连接的信任链，常见的访问入口就是地址栏头儿上的锁图标，点击即可查看（图 4-6）。

- igvita.com 证书由 StartCom Class 1 Primary Intermediate Server 签发。
- StartCom Class 1 Primary Intermediate Server 证书由 StartCom Certification Authority 签发。
- StartCom Certification Authority 被认为是根证书颁发机构。

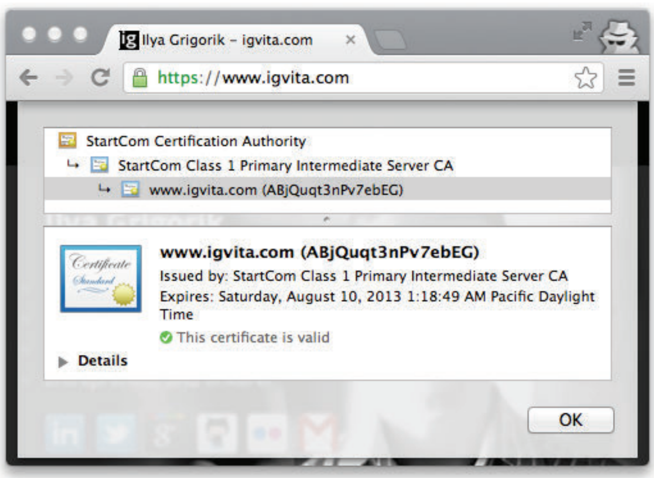


图 4-6：igvita.com 的证书信任链（谷歌 Chrome v25）

整个链条的“信任依据”是根证书颁发机构，在这里就是 StartCom Certification Authority。每个浏览器都会内置一个可信任的证书颁发机构（根机构）的名单，在此浏览器相信而且能够验证 StartCom 根证书。实际上，通过浏览器到浏览器开发商，再到 StartCom 证书颁发机构的信任链传递，可以把信任扩展至目标站点。

> 所有操作系统和浏览器在默认情况下都会提供一个它们信任的证书颁发机构名单。如果你想进一步了解，可以搜索并研究研究这个名单。
>
> 实践中，知名和可信任的证书颁发机构有好几百个，而这也是系统经常遭到责难的原因。想象一下，这么多证书颁发机构无疑会构成一个较大的攻击面，从而给坏人潜入你的浏览器信任链提供可乘之机。

4.5 证书撤销

有时候，出于种种原因，证书颁发者需要撤销或作废证书，比如证书的私钥不再安

全、证书颁发机构本身被冒名顶替，或者其他各种正常的原因，像以旧换新或所属关系更替等。为此，证书本身会包含如何检测其是否过期的指令（图 4-7）。为确保信任链不被破坏，通信的任何一端都可以根据嵌入的指令和签名检查链条中每个证书的状态。

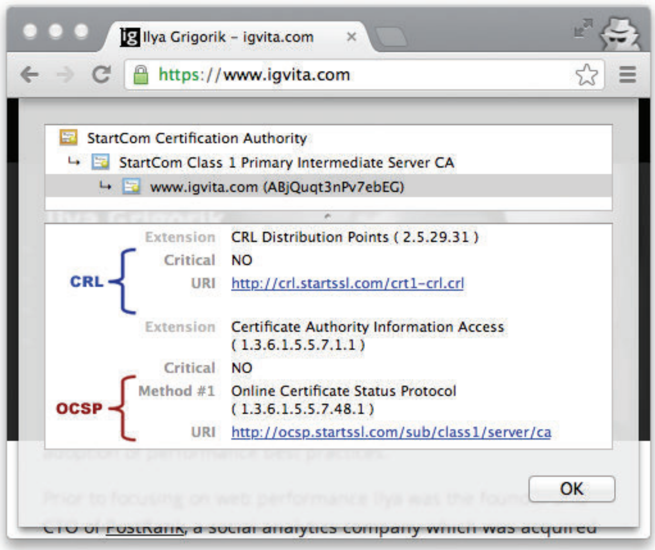


图 4-7：igvita.com 证书中的 CRL 和 OCSP 指令（谷歌 Chrome v25）

4.5.1　证书撤销名单（CRL）

CRL（Certificate Revocation List，证书撤销名单）是 RFC 5280 规定的一种检查所有证书状态的简单机制：每个证书颁发机构维护并定期发布已撤销证书的序列号名单。这样，任何想验证证书的人都可以下载撤销名单，检查相应证书是否榜上有名。如果有，说明证书已经被撤销了。

CRL 文件本身可以定期发布、每次更新时发布，或通过 HTTP 或其他文件传输协议来提供访问。这个名单同样由证书颁发机构签名，通常允许被缓存一定时间。实践中，这种机制效果很好，但也存在一些问题：

- CRL 名单会随着要撤销的证书增多而变长，每个客户端都必须取得包含所有序列号的完整名单；
- 没有办法立即更新刚刚被撤销的证书序列号，比如客户端先缓存了 CRL，之后某证书被撤销，那到缓存过期之前，该证书将一直被视为有效。

4.5.2 在线证书状态协议（OCSP）

为解决 CRL 机制的上述问题，RFC 2560 定义了 OCSP（Online Certificate Status Protocol，在线证书状态协议），提供了一种实时检查证书状态的机制。与 CRL 包含被撤销证书的序列号不同，OCSP 支持验证端直接查询证书数据库中的序列号，从而验证证书链是否有效。总之，OCSP 占用带宽更少，支持实时验证。

然而，没有什么机制是完美无缺的！实时 OCSP 查询也带了一些问题：

- 证书颁发机构必须处理实时查询；
- 证书颁发机构必须确保随时随地可以访问；
- 客户端在进一步协商之前阻塞 OCSP 请求；
- 由于证书颁发机构知道客户端要访问哪个站点，因此实时 OCSP 请求可能会泄露客户端的隐私。

> 实践中，CRL 和 OCSP 机制是互补存在的，大多数证书既提供指令也支持查询。
>
> 更重要的倒是客户端的支持和行为。有的浏览器会分发自己的 CRL 名单，有的浏览器从证书颁发机构取得并缓存 CRL 文件。类似地，有的浏览器会进行实时 OCSP 检查，但在 OCSP 请求失败的情况下行为又会有所不同。要了解具体的情况，可以检查浏览器和操作系统的证书撤销网络设置。

4.6 TLS记录协议

与位于其下的 IP 或 TCP 层没有什么不同，TLS 会话中交换的所有数据同样使用规格明确的协议进行分帧（图 4-8）。TLS 记录协议负责识别不同的消息类型（握手、警告或数据，通过“内容类型”字段），以及每条消息的安全和完整性验证。

字节	+0	+1	+2	+3
0	内容类型			
1..4	版本		长度	
5..n	净荷			
n..m	MAC			
m..p	填充（仅适用于块加密）			

图 4-8：TLS 记录结构

交付应用数据的典型流程如下。

- 记录协议接收应用数据。
- 接收到的数据被切分为块：最大为每条记录 2^{14} 字节，即 16 KB。
- 压缩应用数据（可选）。
- 添加 MAC（Message Authentication Code）或 HMAC。
- 使用商定的加密套件加密数据。

以上几步完成后，加密数据就会被交给 TCP 层传输。接收端的流程相同，顺序相反：使用商定的加密套件解密数据、验证 MAC、提取并把数据转交给上层的应用。

同样，值得庆幸的是以上过程都由 TLS 层帮我们处理，而且对大多数应用都是完全透明的。不过，记录协议也带来了一些重要的限制，务必要注意：

- TLS 记录最大为 16 KB；
- 每条记录包含 5 字节的首部、MAC（在 SSL 3.0、TLS 1.0、TLS 1.1 中最多 20 字节，在 TLS 1.2 中最多 32 字节），如果使用块加密则还有填充；
- 必须接收到整条记录才能开始解密和验证。

有可能的话，应该自主选择记录大小，这也是一项重要的优化。小记录会因记录分帧而招致较大开销，大记录在被 TLS 层处理并交付应用之前，必须通过 TCP 传输和重新组装。

4.7 针对TLS的优化建议

鉴于网络协议的分层架构，在 TLS 之上运行应用与直接通过 TCP 通信没有什么不同。正因为如此，只需要对应用进行很少改动甚至不用改动就可以让它基于 TLS 通信。当然，前提是你根据 2.5 节“针对 TCP 的优化建议”中的最佳实践去做了。

不过，你还应该关心 TLS 部署运维的一些方法，比如服务器的部署方式和地理位置、TLS 记录及内存缓冲区大小、是否支持简短握手，等等。关注这些细节不仅能大大改善用户体验，还能帮你节省运维成本。

4.7.1 计算成本

建立和维护加密信道给两端带来了额外的计算复杂度。特别地，首先有一个非对称（公钥）加密，我们在 4.2 节“TLS 握手”中介绍过。然后，在握手期间确定共享密钥，作为对后续 TLS 记录加密的对称密钥。

如前所述，公钥加密与对称加密相比，需要更大的计算工作量。因此，在 Web 发展早期，通常都需要专门的硬件来进行“SSL 卸载”。好在现在不这样了。现代硬件突

飞猛进的发展为减小这种损失提供了强力支持，原先需要专门硬件来做的工作，今天直接通过 CPU 就能完成。Facebook、谷歌等大公司都通过 TLS 向数百万用户提供服务，相关的计算工作通过软件和普通的计算机就能完成。

> 今年（2010 年）1 月份，Gmail 切换为默认使用 HTTPS。在此之前，它只是一个选项，而现在所有用户都在使用 HTTPS 在浏览器与谷歌之间随时随地安全地收发邮件。为做到这一点，我们并没有部署额外的机器，也没有专用硬件。在我们的前端机器上，SSL/TLS 计算只占 CPU 负载的不到 1%，每个连接只占不到 10 KB 的内存，以及不到 2% 的网络资源。很多人认为 SSL/TLS 占用了很多 CPU 时间，我们希望上述几个数字（首次公开）能消除人们的顾虑。
>
> 如果你现在不想往下再看了，那只需要记住一件事：SSL/TLS 计算已经不是问题了。
>
> ——Adam Langley（谷歌）

> 我们已经大规模部署了 TLS，既使用了硬件也使用软件负载均衡器。我们发现当前基于软件的 TLS 实现在普通 CPU 上已经运行得足够快，无需借助专门的加密硬件就能够处理大量的 HTTPS 请求。我们使用运行于普通硬件上的软件提供所有 HTTPS 服务。
>
> ——Doug Beaver（Facebook）

尽管如此，像 4.3 节“TLS 会话恢复”中介绍的技巧对优化性能依旧很重要，它能帮你减少计算损失和 TLS 握手期间的公钥加密延迟。CPU 也不应该处理本不该处理的计算。

> 说到优化 CPU 周期，一定别忘了把你的 SSL 库升级到最新版本，在此基础上再构建 Web 服务器和代理服务器！最新版的 OpenSSL 在性能方面有了明显的提升，而你系统中默认的 OpenSSL 库很有可能已经过时了。

4.7.2　尽早完成（握手）

建立连接的延迟体现在每个 TLS 连接上，包括新连接和恢复的连接，因此是优化的重点。我们知道 TCP 连接首先要经过 2.1 节描述的“三次握手”，两端要通过一次完整的往返交换 SYN/SYN-ACK 分组。其次，4.2 节介绍的“TLS 握手”在完整的情况下，需要两次额外的往返，或在 4.3 节描述的“TLS 会话恢复”的情况下，需要一次额外的往返。

最差的情况，在实际交换应用数据之前，建立 TCP 和 TLS 连接的过程要经过三次往返！以前面纽约的客户端连接伦敦的服务器为例，每次往返耗时 56 ms（见表 1-1），

那么建立完整的 TCP 和 TLS 连接需要三次往返即 168 ms，而恢复 TLS 会话需要 112 ms。当然，56 ms 是最乐观的数字，正常情况下延迟越长，性能损失越严重。

因为所有 TLS 会话都是在 TCP 之上完成的，因此 2.5 节“针对 TCP 的优化建议”在这里也完全适用。如果说重用 TCP 连接对于非加密通信是一个重要的优化手段，那么这个手段对运行在 TLS 上的应用同样至关重要。换句话说，只要能省掉握手，就应该省掉。如果必须握手，那么还有一个可能的技巧：尽早完成。

第 1 章讨论过，我们不能指望延迟在将来能下降多少，因为光电信号的传输速度已经是光速的一个非常小的常数因子了。不能让分组传播更快，但可以缩短传播距离！尽早完成就是这么一个技巧，即通过把服务器放到离用户更近的地方（图 4-9），让客户端与服务器之间往返延迟最少。

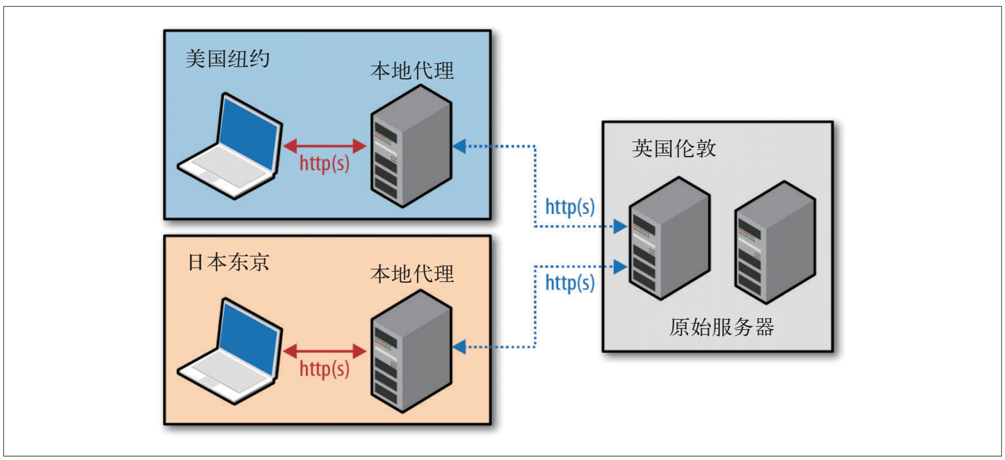


图 4-9：客户端连接的尽早完成

做到尽早完成的最简单方式，就是在世界各地的服务器上缓存或重复部署数据和服务，而不要让所有用户都通过跨海或跨大陆光缆连接到一个中心原始服务器。当然，这正是 CDN（Content Delivery Networks，内容分发网络）服务的内容：通过使用本地代理服务器分流负载等手段降低延迟。

虽然 CDN 最常用于在全球优化分发静态资源，但其优点并不止于此。距离客户端更近的服务器还可以缩短 TLS 会话，因为 TCP 和 TLS 握手的对象都是近处的服务器，所以建立连接的总延迟就会显著减少。相应地，本地代理服务器则可以与原始服务器建立一批长期的安全连接，全权代理请求与响应。

简言之，把服务器放到接近客户端的地方能够节约 TCP 和 TLS 握手的时间！大多数 CDN 提供商都提供这种服务，当然如果你愿意也可以以最小的成本部署自己的基础

设施：在全球几大数据中心租用一些云服务器，然后在每台服务器上都运行代理（服务器）程序，把请求转发到你的原始服务器，再加上地理 DNS 负载均衡系统即可。

> **不缓存的原始获取**
>
> 使用 CDN 或代理服务器取得资源的技术，如果要根据用户定制或者涉及隐私数据，则不能做到全球缓存，这种情况被称为“不缓存的原始获取”（uncached origin fetch）。
>
> 虽然只有把数据缓存到全球各地的服务器上 CDN 才能发挥最大的效用，但“不缓存的原始获取”仍然具有性能优势：客户端连接终止于附近的服务器，从而显著减少握手延迟。相应地，CDN 或你的代理服务器可以维护一个“热连接池”（warm connection pool），通过它将数据转发给原始服务器，同时做到对客户端快速响应。
>
> 事实上，作为附加的一个优化层，CDN 提供商在连接两端都会使用邻近服务器！客户端连接终止于邻近 CDN 节点，该节点将请求转发到与对端服务器邻近的 CDN 节点，之后请求才会被路由到原始服务器。CDN 网络中多出来这一跳，可以让数据在优化的 CDN 骨干网中寻路，从而进一步减少客户端与服务器之间的延迟。

4.7.3　会话缓存与无状态恢复

无论什么情况下，在接近用户的地方终止连接都有助于减少延迟，但有延迟终归快不过没有延迟。启用 TLS 会话缓存和无状态恢复可以完全消除“回头客”的往返时间。

SSL 2.0 引入的会话标识符机制是 TLS 会话缓存的基础，目前已经得到大多数客户端和浏览器的广泛支持。然而，如果你在自己的服务器上配置 SSL/TLS，千万不能主观认为该机制默认是开启的。实际上，在大多数服务器的默认配置下它是禁用的。为此，应该仔细检查和验证自己的配置：

- 支持多进程或工作进程的服务器应该使用共享的会话缓存；
- 共享的会话缓存的大小应该根据流量调整；
- 应该设置会话超时时间；
- 在多台服务器并存的情况下，把相同的客户端 IP 或相同的 TLS 会话 ID 路由到同一台服务器可以最好地利用会话缓存；
- 在不适宜使用“单一”负载均衡策略的情况下，应该为多台服务器配置共享缓存，以便最好地利用会话缓存；
- 检查和监控 SSL/TLS 会话缓存的使用情况，以之作为性能调优的依据。

此外，如果客户端和服务器都支持会话记录单，则所有会话数据都将保存在客户端，上述步骤就都不需要了（问题一下子就简单了很多）！不过，由于会话记录单还是相对新的 TLS 扩展，并非所有客户端都支持它。实践中，为了取得最优结果，应该做好两手准备：在支持的客户端中使用会话记录单，而在不支持的客户端中使用会话标识符。这两种手段不会相互干扰，而是会很好地协同工作。

4.7.4 TLS记录大小

所有通过 TLS 交付的应用数据都会根据记录协议传输（图 4-8）。每条记录的上限为 16 KB，视选择的加密方式不同，每条记录还可能额外带有 20 到 40 字节的首部、MAC 及可选的填充信息。如果记录可以封装在一个 TCP 分组内，则还会给它增加相应的 IP 和 TCP 字段，即 20 字节的 IP 首部和 20 字节的 TCP 首部（无选项）。结果，每条记录的大小就变成了 60 到 100 字节。由于 MTU 通常为 1500 字节，因此分组占比最小的情况下只相当于帧大小的 6%。

记录越小，分帧浪费越大。但简单地把记录增大到上限（60 KB）也不一定好！如果记录要分成多个 TCP 分组，那 TLS 层必须等到所有 TCP 分组都到达之后才能解密数据（图 4-10）。只要有 TCP 分组因拥塞控制而丢失、失序或被节流，那就必须将相应 TLS 记录的分段缓存起来，从而导致额外的延迟。实践中，这种延迟会造成浏览器性能显著下降，因为浏览器倾向于逐字节地读取数据。

```
▽ [8 Reassembled TCP Segments (11221 bytes): #169(1460), #170(1460), #172(1460), #174(1460),
                                             #175(1460), #177(1460), #179(1460), #180(1001)]
    [Frame: 169, payload: 0-1459 (1460 bytes)]
    [Frame: 170, payload: 1460-2919 (1460 bytes)]
    [Frame: 172, payload: 2920-4379 (1460 bytes)]
    [Frame: 174, payload: 4380-5839 (1460 bytes)]
    [Frame: 175, payload: 5840-7299 (1460 bytes)]
    [Frame: 177, payload: 7300-8759 (1460 bytes)]
    [Frame: 179, payload: 8760-10219 (1460 bytes)]
    [Frame: 180, payload: 10220-11220 (1001 bytes)]
    [Segment count: 8]
    [Reassembled TCP length: 11221]
▽ Secure Sockets Layer
  ▽ TLSv1 Record Layer: Application Data Protocol: http
      Content Type: Application Data (23)
      Version: TLS 1.0 (0x0301)
      Length: 11216
      Encrypted Application Data: 07ed92e420530da2e2755a5b5372ef32b53e0d4e7c20c3d8...
```

图 4-10：WireShark 的截图，其中 11 211 字节的 TLS 记录被分成了 8 个 TCP 段

小记录会造成浪费，大记录会导致延迟。因此，记录到底多大合适没有唯一“正确”的答案。不过对于在浏览器中运行的 Web 应用来说，倒是有一个值得推荐的做法：每个 TCP 分组恰好封装一个 TLS 记录，而 TLS 记录大小恰好占满 TCP 分配的

MSS（Maximum Segment Size，最大段大小）。换句话说，一方面不要让 TLS 记录分成多个 TCP 分组，另一方面又要尽量在一条记录中多发送数据。以下数据可作为确定最优 TLS 记录大小的参考：

- IPv4 帧需要 20 字节，IPv6 需要 40 字节；
- TCP 帧需要 20 字节；
- TCP 选项需要 40 字节（时间戳、SACK 等）。

假设常见的 MTU 为 1500 字节，则 TLS 记录大小在 IPv4 下是 1420 字节，在 IPv6 下是 1400 字节。为确保向前兼容，建议使用 IPv6 下的大小：1400 字节。当然，如果 MTU 更小，这个值也要相应调小。

可惜的是，我们不能在应用层控制 TLS 记录大小。TLS 记录大小通常是一个设置，甚至是 TLS 服务器上的编译时常量或标志。要了解具体如何设置这个值，请参考服务器文档。

> 如果服务器要处理大量 TLS 连接，那么关键的优化是把每个连接占用的内存量控制在最小。默认情况下，OpenSSL 等常用的库会给每个连接分配 50 KB 空间，但正像设置记录大小一样，有必要查一查文档或者源代码，然后再决定如何调整这个值。谷歌的服务器把 OpenSSL 缓冲区的大小减少到了大约 5 KB。

4.7.5 TLS压缩

TLS 还有一个内置的小功能，就是支持对记录协议传输的数据进行无损压缩。压缩算法在 TLS 握手期间商定，压缩操作在对记录加密之前执行。然而，出于如下原因，实践中往往需要禁用服务器上的 TLS 压缩功能：

- 2012 年公布的“CRIME”攻击会利用 TLS 压缩恢复加密认证 cookie，让攻击者实施会话劫持；
- 传输级的 TLS 压缩不关心内容，可能会再次压缩已经压缩过的数据（图像、视频，等等）。

双重压缩会浪费服务器和客户端的 CPU 时间，而且暴露的安全漏洞也很严重，因此请禁用 TLS 压缩。实践中，大多数浏览器会禁用 TLS 压缩，但即便如此你也应该在服务器的配置中明确禁用它，以保护用户的利益。

> 虽然不能使用 TLS 压缩，但应该使用服务器的 Gzip 设置压缩所有文本资源，同时对图像、视频、音频等媒体采用最合适的压缩格式。

4.7.6 证书链的长度

验证信任链需要浏览器遍历链条中的每个节点，从站点证书开始递归验证父证书，直至信任的根证书。因此，优化的首要工作就是检查服务器在握手时没有忘记包含所有中间证书。如果忘记了包含中间证书，虽然很多浏览器可以正常工作，但它们会暂停验证并自己获取中间证书，验证之后再继续。此时很可能需要进行新 DNS 查找、建立 TCP 连接、发送 HTTP GET 请求，导致握手多花几百 ms 时间。

> 浏览器怎么知道到哪里去找证书呢？子证书中通常包含父证书的 URL。

另一方面，还要确保信任链中不包含不必要的证书！或更一般化地讲，应该确保证书链的长度最小。我们在 4.2 节“TLS 握手”中介绍过，服务器证书是在握手期间发送的，而发送证书使用的很可能是一个处于 2.2.2 节“慢启动”算法初始阶段的新 TCP 连接。如果证书链长度超过了 TCP 的初始拥塞窗口（图 4-11），那我们无意间就会让握手多了一次往返：证书长度超过拥塞窗口，从而导致服务器停下来等待客户端的 ACK 消息。

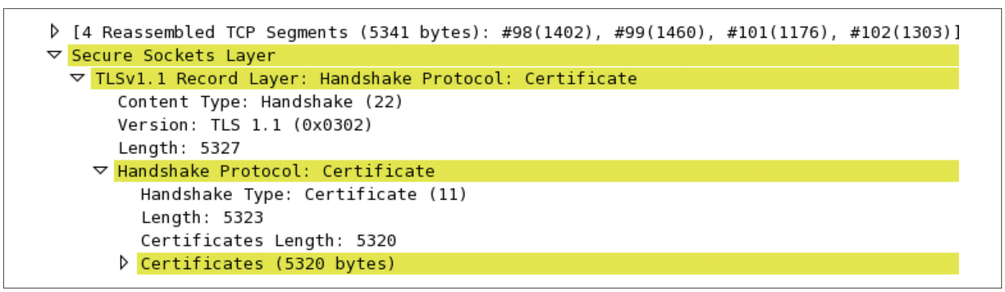


图 4-11：WireShark 的截图显示的 5323 字节的 TLS 证书链

图 4-11 所示的证书链超过了 5 KB，也超过了大多数旧版本浏览器的初始拥塞窗口大小，因此会在握手期间增加一次额外的往返。对此，可以通过增大拥塞窗口来解决，参见 2.2.2 节最后的“增大 TCP 的初始拥塞窗口”部分。此外，就是要看一看能否减少要发送的证书大小了。

- 尽量减少中间证书颁发机构的数量。理想情况下，发送的证书链应该只包含两个证书：站点证书和中间证书颁发机构的证书。把这一条作为选择证书颁发机构的标准。第三个证书，也就是根证书颁发机构的证书，已经包含在浏览器内置的信任名单中了，不用发送。
- 很多站点会在证书链中包含根证书颁发机构的证书，这是完全没有必要的。如果浏览器的信任名单中没有该根证书，那说明它是不被信任的，即便发送它也无济于事。

- 理想的证书链应该在 2 KB 或 3 KB 左右，同时还能给浏览器提供所有必要的信息，避免不必要的往返或者对证书本身额外的请求。优化 TLS 握手可以消除关键的性能瓶颈，因为每个新 TLS 连接都要经历同样的延迟。

4.7.7 OCSP封套

每个新 TLS 连接都要求浏览器验证发送过来的证书链的签名。然而，不要忘了还有一步：浏览器也需要验证证书没有被撤销。为此，浏览器可能会定期下载并缓存 4.5.1 节提到的证书颁发机构发布的“证书撤销名单（CRL）”，而且还可能需要分派发送一个 4.5.2 节提到的“在线证书状态协议（OCSP）”请求，以便实时验证。遗憾的是，浏览器在这时候的行为差别很大。

- 某些浏览器会使用自己的更新机制推送更新的 CRL 名单，而不会按需发送请求。
- 某些浏览器可能只会针对扩展验证证书（EV 证书）进行实时 OCSP 和 CRL 检查。
- 某些浏览器可能会在上述任何一种方式下阻塞 TLS 握手，有些则不会，具体情况取决于开发商、平台和浏览器版本。

这里的情况很复杂，也没有最好的解决方案。不过，在某些浏览器中还是可以采用一个叫做 OCSP 封套（OCSP stapling）的优化措施：服务器可以在证书链中包含（封套）证书颁发机构的 OCSP 响应，让浏览器跳过在线查询。把查询 OCSP 操作转移到服务器可以让服务器缓存签名的 OCSP 响应，从而节省很多客户端的请求。与此同时，还要注意一些情况。

- OCSP 响应从 400 字节到 4000 字节不等。把这么大的响应封套在证书链里照样会造成 TCP 拥塞窗口溢出，因此要关注整体大小。
- 只能包含一个 OCSP 响应，即在没有缓存的情况下，浏览器对其他中间证书可能仍然需要发送 OCSP 请求。

最后，要启用 OCSP 封套，还需要服务器支持才行。好在，NginX、Apache 和 IIS 等服务器都可以通过配置支持 OCSP 封套。对于其他服务器，请参考文档中的说明。

4.7.8 HTTP严格传输安全（HSTS）

HTTP 严格传输安全（HSTS，Strict Transport Security）是一种安全策略机制，它能让服务器通过简单的 HTTP 首部（如 `Strict-Transport-Security: max-age=31536000`）对适用的浏览器声明访问规则。具体来讲，它可以让用户代理遵从如下规则：

- 所有对原始服务器的请求都通过 HTTPS 发送；
- 所有不安全的链接和客户端请求在发送之前都应该在客户端自动转换为 HTTPS；

- 万一证书有错误，则显示错误消息，用户不能回避警告；
- `max-age` 以秒为单位指定 HSTS 规则集的生存时间（例如，`max-age=31536000` 等于缓存 365 天）；
- 用户代理可以根据指令在指定的证书链中记住（“印下”）某主机的指纹，以便将来访问时使用，从而有效限制证书颁发机构在特定时间（由 `max-age` 指定）内可颁发证书的范围。（这一项是可选的。）

事实上，HSTS 会把原始服务器转换为只处理 HTTPS 的目标服务器，从而确保应用不会因各种主动或被动攻击给用户造成损失。从性能角度说，HSTS 通过把责任转移到客户端，让客户端自动把所有链接重写为 HTTPS，消除了从 HTTP 到 HTTPS 的重定向损失。

> 到 2013 年初，支持 HSTS 的浏览器有 Firefox 4+、Chrome 4+、Opera 12+ 和 Android 平台的 Chrome 和 Firefox。要了解最新的支持情况，请访问：http://caniuse.com/stricttransportsecurity。

4.8 性能检查清单

作为应用开发人员，事实上你接触不到 TLS 的这些复杂性。只要别把页面中的 HTTP 和 HTTPS 内容混为一谈，那你的应用就可以在这两种情况下都顺畅运行。然而，应用的整体性能却会受到服务器底层配置的影响。

好在，只要想到优化，任何时候都不算晚。而且一旦优化到位，所有与服务器的新连接都将受益无穷！下面是一个简单的检查清单：

- 要最大限制提升 TCP 性能，请参考 2.5 节“针对 TCP 的优化建议”；
- 把 TLS 库升级到最新版本，在此基础上构建（或重新构建）服务器；
- 启用并配置会话缓存和无状态恢复；
- 监控会话缓存的使用情况并作出相应调整；
- 在接近用户的地方完成 TLS 会话，尽量减少往返延迟；
- 配置 TLS 记录大小，使其恰好能封装在一个 TCP 段内；
- 确保证书链不会超过拥塞窗口的大小；
- 从信任链中去掉不必要的证书，减少链条层次；
- 禁用服务器的 TLS 压缩功能；
- 启用服务器对 SNI 的支持；
- 启用服务器的 OCSP 封套功能；
- 追加 HTTP 严格传输安全首部。

4.9 测试与验证

最后，要验证和测试你的配置，可以使用 Qualys SSL Server Test（https://www.ssllabs.com/ssltest/）等在线服务来扫描你的服务器，以发现常见的配置和安全漏洞。此外，最好熟练掌握 openssl 命令行工具，通过它来检查整个握手和本地服务器配置情况：

```
$> openssl s_client -state -CAfile startssl.ca.crt -connect igvita.com:443

  CONNECTED(00000003)
  SSL_connect:before/connect initialization
  SSL_connect:SSLv2/v3 write client hello A
  SSL_connect:SSLv3 read server hello A
  depth=2 /C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
          /CN=StartCom Certification Authority
  verify return:1
  depth=1 /C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
          /CN=StartCom Class 1 Primary Intermediate Server CA
  verify return:1
  depth=0 /description=ABjQuqt3nPv7ebEG/C=US
          /CN=www.igvita.com/emailAddress=ilya@igvita.com
  verify return:1
  SSL_connect:SSLv3 read server certificate A
  SSL_connect:SSLv3 read server done A ❶
  SSL_connect:SSLv3 write client key exchange A
  SSL_connect:SSLv3 write change cipher spec A
  SSL_connect:SSLv3 write finished A
  SSL_connect:SSLv3 flush data
  SSL_connect:SSLv3 read finished A
  ---
  Certificate chain ❷
   0 s:/description=ABjQuqt3nPv7ebEG/C=US
       /CN=www.igvita.com/emailAddress=ilya@igvita.com
     i:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
       /CN=StartCom Class 1 Primary Intermediate Server CA
   1 s:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
       /CN=StartCom Class 1 Primary Intermediate Server CA
     i:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
       /CN=StartCom Certification Authority
  ---
  Server certificate
  -----BEGIN CERTIFICATE-----
  ... snip ...
  ---
  No client certificate CA names sent
  ---
  SSL handshake has read 3571 bytes and written 444 bytes ❸
  ---
  New, TLSv1/SSLv3, Cipher is RC4-SHA
  Server public key is 2048 bit
  Secure Renegotiation IS supported
  Compression: NONE
```

```
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : RC4-SHA
    Session-ID: 269349C84A4702EFA7 ... ❹
    Session-ID-ctx:
    Master-Key: 1F5F5F33D50BE6228A ...
    Key-Arg   : None
    Start Time: 1354037095
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

❶ 客户端完成对接收到的证书链的验证

❷ 接收到的证书链（2 个证书）

❸ 接收到证书链的大小

❹ 对有状态 TLS 恢复发送的会话标识符

在上面的例子中，我们连接到 igvita.com 默认的 TLS 端口（443），并进行了 TLS 握手。因为 `s_client` 假设没有根证书，所以我们手工把路径指定为 StartSSL Certificate Authority 的根证书，这一点很重要。你的浏览器内置了 StartSSL 的根证书，因此可以验证这个信任链，但 `s_client` 没有依赖于此。如果在这里忽略根证书，应该会在日志中看到验证错误。

通过检查证书链，我们发现服务器发送了两个证书，累计起来是 3571 字节，差不多相当于 3~4 个 TCP 初始拥塞窗口的大小。这里要注意不能超过拥塞窗口大小，否则就要考虑增大服务器上 cwnd 的值。最后，我们可以看到协商后的 SSL 会话变量：选择的协议、加密套件、密钥等。还可以看到服务器为当前会话发送的会话标识符，这个标识符在将来恢复时会用到。

第二部分

无线网络性能

第 5 章

无线网络概览

5.1 无所不在的连接

过去十年来，最具颠覆性的技术趋势非随时随地上网莫属，人们对随时随地上网的需求也与日俱增。无论是查收邮件、语音聊天、上网浏览，还是其他需求，总之人们都希望随时随地访问各种在线服务：跑步时、排队时、上班时、乘地铁时、坐飞机时……不受地域、时间及环境的限制。今天，为满足这些需求，我们经常还是要提前找接入点（比如寻找附近的 Wi-Fi 热点），但毋庸置疑，未来的愿景一定是随时随地上网。

无线网络是这股趋势的核心所在。宽泛地说，无线网络可以指任何不通过线缆连接的网络，只有不通过线缆才能满足移动用户随时随地上网的需求。毫不奇怪，面对那么多的使用场景和应用形式，无线技术自然也会多种多样，而且各有各的特点，各有各的优势。Wi-Fi、蓝牙、ZigBee、NFC、WiMAX、LTE、HSPA、EVDO，以及早先的 3G 标准、卫星服务等，都是我们今天司空见惯的无线网络技术。

鉴于无线网络技术如此多样，笼统地概括所有无线网络的性能优化手段是不可能的。然而，好在大多数无线技术的原理都是相通的，各种技术也都具有类似的权衡因素，而且衡量性能的指标和约束条件也具有普遍适用性。只要我们把影响无线性能的基本原理搞清楚，那其他问题自然也就迎刃而解了。

另外，尽管通过无线电传播数据与通过有线传输数据有着本质的差别，但用户期待的结果都是（或者说应该是）一样的，一样的性能，一样的结果。从长远来看，所

有应用都会或都将会通过无线网络交付，只不过某些应用通过无线网络来获取的步伐会走得更快一些。世上根本不存在什么*有线应用*，交付方式是有线还是无线，用户才不关心呢。

无论连接方式是有线还是无线，应用都应该一如既往地正常运行。从用户角度说，他不用考虑底层技术，但作为开发人员，则必须在设计阶段就考虑如何在应用中弥合不同网络的差异。好消息是，我们用来优化无线网络的各种手段，在其他场景下也能带来更好的用户体验。下面我们就开始吧。

5.2 无线网络的类型

网络由一组设备互相连接构成。对无线网络而言，无线电是通信媒体。可是，同为无线电媒体，应用范围不同、拓扑结构不同、使用场景不同，相应的网络技术也迥然不同。下面，我们就从地理范围角度对无线网络技术加以分类（表 5-1）。

表5-1：无线网络的类型

类型	范围	应用	标准
个人局域网（PAN）	个人活动范围内	替代周边设备的线缆	蓝牙、ZigBee、NFC
局域网（LAN）	一栋楼或校园内	有线网络的无线扩展	IEEE 802.11（Wi-Fi）
城域网（MAN）	一座城市内	无线内联网	IEEE 802.15（WiMax）
广域网（WAN）	世界范围内	无线网络	蜂窝（UMTS、LTE 等）

这种分类方式既不全面，也不精确。很多技术和标准最初都源自某个特定的使用场景，例如蓝牙技术最初就是针对 PAN 应用和为替代线缆而出现的。这些技术随着时间推移功能不断增强，应用范围和吞吐量也不断扩大。比如，蓝牙技术现在就支持在高带宽使用场景下与 802.11（Wi-Fi）的无缝互操作。类似地，WiMAX 最初的应用场景是定点无线连接，但随着时间推移也具备了移动能力，从而成为了其他 WAN 或蜂窝技术的替代方案。

上述分类方式的关键不在于怎么分清各种技术之间的界限，而在于突出每种使用场景下的宏观差异。有些设备的供电可以连绵不绝，而其他设备则需要随时考虑节省电量。有些需要 Gbit/s 级的传输速率，而有些生来只能传输几十到几百字节的数据（如 NFC）。有些应用需要实时在线，而其他应用则可以容许延迟。正是包含这些在内的很多因素决定了每种网络的最初特点。然而，每种网络技术一旦付诸应用，就会进一步发展。电池容量越来越大，处理器速度越来越快，调制算法越来越先进，加上其他方面的改进，使得每种无线标准的应用场景和性能都得到了扩展。

你的下一个应用很可能要通过移动网络交付，而它同时还可能要通过 NFC 付款、通过蓝牙实现基于 WebRTC 的 P2P 通信、通过 Wi-Fi 传输高清视频流。总之，无线应用不可能局限于一种无线标准！

5.3 无线网络的性能基础

所有无线技术都有自身的约束和局限。然而，无论使用哪种无线技术，所有通信方法都有一个最大的信道容量，这个容量是由相同的底层原理决定的。事实上，信息论之父克劳德 · 香农对此给出了一个确切的数学模型（公式 5-1），这个模型与技术无关。

公式 5-1：信道容量即最大信息速率

$$C = BW \times \log_2 \left(1 + \frac{S}{N}\right)$$

- C 是信道容量，单位是 bit/s；
- BW 是可用带宽，单位是 Hz；
- S 是信号，N 是噪声，单位是 W。

尽管存在某种程度的简化，但这个公式涵盖了影响大多数无线网络性能的所有基本因素。在所有这些因素中，与数据传输速度最直接相关的就是接收端与发送端之间的可用带宽和信号强度。

5.3.1 带宽

有线网络通过线缆将网络中的各个节点连接起来，而无线电通信本质上则是一个共享媒体，它靠的是无线电波，或者专业一点讲，叫做电磁辐射。为实现通信，发送端与接收端必须事先就通信使用的频率范围达成共识，在这个频率范围内双方可以顺畅地交换信息。比如，802.11b 和 802.11g 在所有 Wi-Fi 设备上都使用 2.4~2.5 GHz 频带。

那么这些频率范围的确定与分配由谁负责呢？简单来说，由地方政府负责（见图 5-1）。在美国，频带分配权掌握在 FCC（Federal Communication Commission，联邦通信委员会）手中。事实上，由于政府规定不同，有些无线技术可以在世界上某些国家和地区使用，但在另一些国家和地区就无法使用。另外，不同国家也经常会给相同的无线技术分配不同的频率范围。

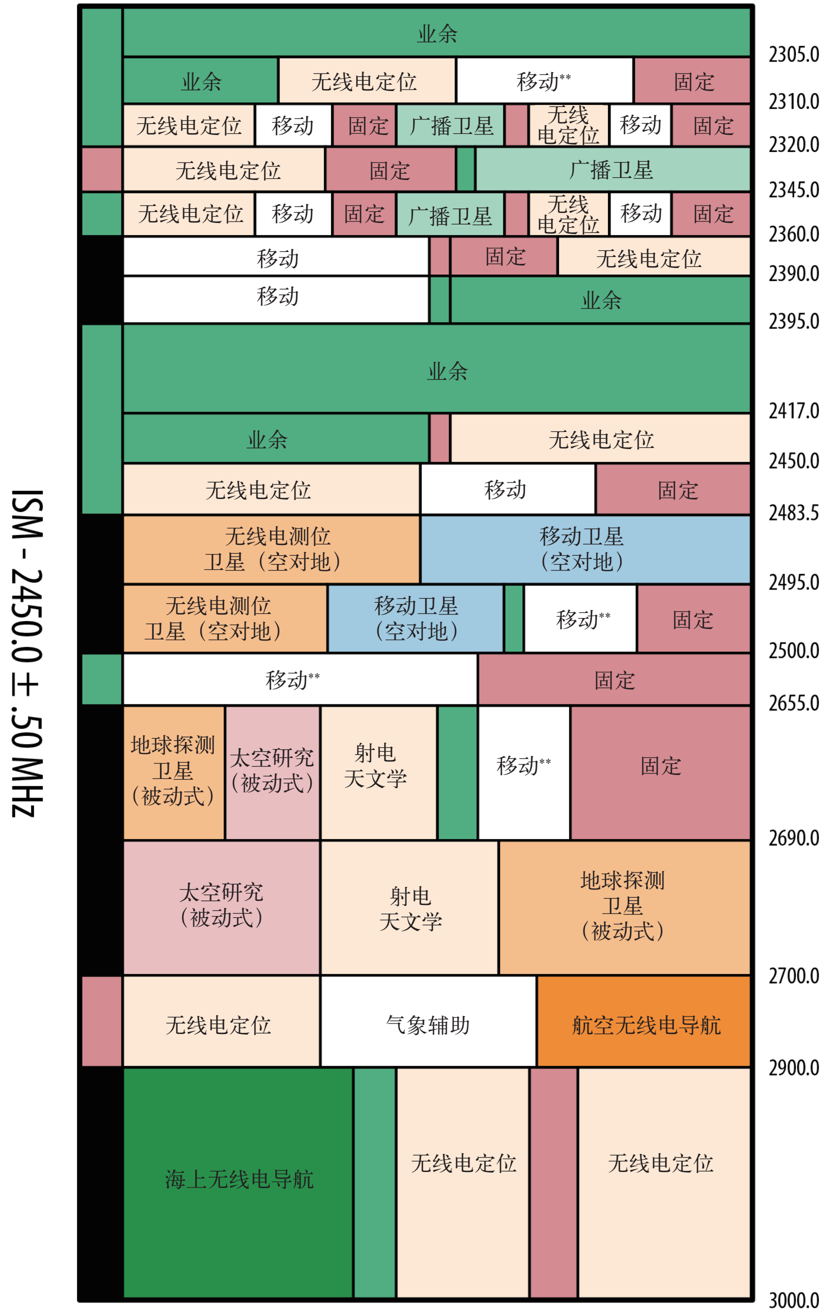


图 5-1：FCC 无线频谱分配图：2300~3000MHz 频段

抛开政治因素不谈，除了以共有频段作为信息交互的基础外，影响性能的最主要因素就是频率范围的大小（带宽）。根据香农的模型（公式 5-1），信道的总体比特率与分配的带宽呈正比。换句话说，在其他条件等同的情况下，频率范围加倍，传输速度加倍。比如，带宽从 20MHz 变成 40MHz 可以让信道传输速率加倍，而这正是 802.11n 在早期 Wi-Fi 标准基础上提升性能的做法！

最后，值得一提的是，并非所有频率范围的性能都一样。低频信号传输距离远、覆盖范围广（大蜂窝），但要求天线更大，而且竞争激烈。另一方面，高频信号能够传输更多数据，但传输距离不远，因此覆盖范围小（小蜂窝），需要较多的基础设施投入。

> 对不同的应用而言，不同的频率范围价值也不一样。广播只适合低频率范围，而双向通信则更适合使用较小的蜂窝，因为较小的蜂窝能提供更高的带宽，竞争也更少。

全球频谱分配及管制简史

只要对世界无线通信史稍有涉猎，就不可避免地会了解一些当前频谱分配和管制方面的纷争。那么无线电通信的历史如何呢？

在无线电产业发展初期，任何人可以出于任何目的使用任意频率范围。直到 1912 年美国的《无线电法》签署生效，无线电频谱才揭开强制许可使用的序幕。该法案最初的动议部分源自对泰坦尼克号沉没的调查。据称，如果当时附近的船只都在监听适当的频率，那么就有可能避免灾难发生，至少可以让更多人获救。无论如何，这个新法案开了为国际和美国联邦无线通信立法的先河。此后，其他国家也纷纷效仿。

20 多年后，FCC 出台了 1934 年《通信法》。自此之后，FCC 就一直负责美国境内的频谱分配，将可用频谱分割成了较小的专用频段。

ISM（Industrial, Scientific, and Medical，工业、科研和医疗）无线电频段就是一个差异化分配的例子。这个频段是 1947 年国际电信大会确定的，在国际上得到了保留。现代无线通信（如 Wi-Fi）使用的 2.4~2.5 GHz（100 MHz）和 5.725~5.875 GHz（150 MHz）就属于 ISM 频段。而且，这两个频段也被视为“免许可频段”，即任何人都可以在这两个频段组建自己的无线网络（无论商用还是私用），只要使用的硬件达到一定的技术要求（比如发射功率）即可。

随着无线通信需求日益增长，很多政府都开始举行“频谱拍卖”，通过这种形式出售在某个频段发射信号的许可（执照）。这种例子非常多，但 2008 年的 700MHz FCC 拍卖比较有代表性。当时，美国境内 698~806 MHz 的频率范围被以 195.92 亿美元的价格拍卖给十几个竞拍者（这个范围被进一步细分）。没错，的确是“百”亿级的价格。

带宽是一种稀缺、昂贵的资源。关于当前分配方式是否公平的讨论不绝于耳，也有很多相关的专著出版。放眼未来，有一件事是确定无疑的：这种争论还将持续升温。

5.3.2 信号强度

除了带宽之外，无线通信的第二个限制因素就是收发两端之间的信号强度，也叫信噪比（SNR，Signal Noise Ratio）。本质上，信噪比衡量的是预期信号强度与背景噪声及干扰之间的比值。背景噪声越大，携带信息的信号就必须越强。

本质上，所有无线电通信使用的都是共享媒体，因此别的设备很可能在这个媒体中产生干扰信号。比如，以 2.5 GHz 频率工作的微波炉很可能与 Wi-Fi 使用的频率范围重合，从而产生交叉干扰。此外，邻居家的 Wi-Fi 热点，甚至同事访问同一 Wi-Fi 网络的笔记本电脑，同样可能对你的数据传输产生干扰。

理想情况下，你应该是某一频率范围内的唯一用户，而且也没有背景噪声和干扰。可惜的是，这是不现实的。首先，带宽很稀缺，其次，要实现无线通信必须有很多设备参与。因此，如果想在存在干扰的情况下达到预期的数据传输速度，要么增大发射功率，也就是提高信号强度，要么缩短收发两端的距离——或者双管齐下。

> 路径损耗，或叫通路衰减，指的是信号强度随距离降低。实际的降低速率因环境而异。关于这方面的讨论超出了本书范围，如果你好奇的话，可以自己搜索一下。

要理解信号、噪声、发射功能和距离之间的关系，可以想象你在一个小房间里跟距离你 6 米远的一个人说话。如果房间里只有你们俩人，你们只要用正常音量讲话即可。但是，现在假设房间里又来了二十几个人，就像一个拥挤的聚会，每个人都在跟另一个人说话。突然之间，你发现自己根本听不清对方的声音了！当然，你可以提高音量，但这样一来会增大你周围每个人的“噪声”。于是，他们也会相应地提高音量，从而进一步加大了噪声和干扰。而在知道对方在哪里之前，房间里的每个人只能与距离自己 1 米左右的人对话（图 5-2）。假如你曾在吵闹的聚会上怎么喊也没办法让别人听见，或者曾弯下腰去跟人大声说话，那你就该知道什么是信噪比了。

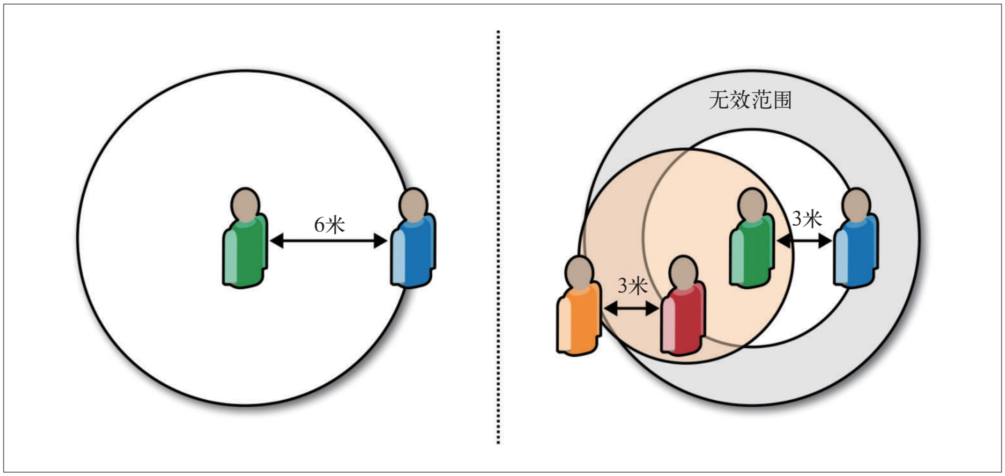


图 5-2：日常生活中小区（cell）呼吸及远近效应

事实上，上面的例子演示了两个重要的效应。

- 远近效应

  接收端捕获较强的信号，因而不可能检测到较弱的信号，实际上是“挤出”了较弱的信号。

- 小区呼吸效应

  小区覆盖范围或信号传输距离基于噪声大小和干扰级别扩展和收缩。

你旁边一个或多个大声讲话的人会阻挡较弱的信号，从而产生远近效应。类似地，你周围交谈的人越多，干扰就越严重，能让你识别有用信号的范围也越小，这就是呼吸效应。由此及彼，就不难理解各种无线电通信中也照样存在与此类似的限制了，而且这种限制与使用什么通信协议和底层技术无关。

5.3.3 调制

可用带宽和 SNR 是影响任何无线信道容量的两个主要物理因素。可是，用于编码信号的算法对无线性能同样有显著影响。

简单来说，我们数字信号（1 和 0）需要转换成模拟信号（无线电波）。调制指的就是这个数模转换过程，不同调制算法的转换效率是不一样的。不同的字母数字组合与符号率决定了信道的最终吞吐量。下面来看一个例子：

- 接收端和发送端每秒可以处理 1000 个脉冲或符号（1000 波特）；
- 传送的每个符号代表一个不同的位序列，由选择的字母数字决定（例如，2 位序列：

00、01、10、11）；
- 信道的比特率是 1000 波特 ×2 比特 / 符号，即每秒 2000 比特。

选择调制算法取决于可用的技术、收发两端的计算能力，以及 SNR。高阶调制的代价是针对噪声和干扰的可靠性降低。天下没有免费的午餐！

> 不要担心，我们不会一头扎进信号处理的泥淖。这里关键是要理解调制算法确实会影响无线信道的容量，但同时它也会受到 SNR、可用处理能力，以及其他常见制约因素的影响。

5.4　测量现实中的无线性能

我们关于信号理论的简明教程可以总结如下：任何无线网络，无论它叫什么名字，缩写是什么或者修订版本是多少，其性能归根结底都受限于几个众所周知的因素。特别是分配给它的带宽大小和收发两端的信噪比。另外，所有利用无线电的通信都：

- 通过共享的通信媒体（无线电波）实现；
- 在管制下使用特定频率范围；
- 在管制下使用特定的发射功率；
- 受限于不断变化的背景噪声和干扰；
- 受限于所选无线技术的技术约束；
- 受限于设备本身的限制，比如形状、电源，等等。

所有无线技术都在宣传自己的峰值或最大数据速率。比如，802.11g 标准的最大传输速率为 54 Mbit/s，而 802.11n 标准则提高到了 600 Mbit/s。类似地，某些移动运营商宣传自己能通过 LTE 提供 100+ MBit/s 的上网速度。然而，面对这些数字，必须知道的是，它们都是在理想条件下的结果。

什么是理想条件？没错，带宽最大、频段独有、噪声最小、调制算法最优，以及日益普及的多无线流并行发射（MIMO，Multiple Input Multiple Output，多输入多输出）。毋庸讳言，使用无线网络与使用有线网络的体验可能（或者说一定）相差极大。

以下仅列举几个影响无线网络性能的因素：

- 收发端的距离；
- 当前位置的背景噪声大小；
- 来自同一网络（小区）其他用户的干扰大小；
- 来自相邻网络（小区）其他用户的干扰大小；

- 两端发射功率大小；
- 处理能力及调制算法。

换句话说，如果你想得到最大的吞吐量，就要尽力减少可控的噪声和干扰，尽量缩短接收端与发射端的距离，给它们必要的功率，并确保两端都选择最优的调制方法。否则，如果你对性能要求十分苛刻，干脆还是使用有线吧！无线的便捷总得有点代价。

没错，度量无线网络的性能并不容易。一个小小的变化，比如把接收端的位置挪动十几厘米，传输速率很可能就会翻倍。而过一小会儿，随着其他接收设备被唤醒并竞争性地访问同一个无线信道，很可能又会导致速率减半。本质上，无线性能具有高度不稳定性。

最后请大家注意，以上讨论关注的一直是吞吐量。这是不是说延迟不重要了？当然不是，但延迟与具体的无线技术密切相关，我们接下来分别讨论。

第 6 章

Wi-Fi

Wi-Fi 工作于免许可的 ISM 频段，任何人在任何地方都可以轻易部署，必要的硬件也很便宜很简单。毫不奇怪，Wi-Fi 已经成为如今最流行也是应用最为广泛的无线技术。

Wi-Fi 是 Wi-Fi 联盟（Wi-Fi Alliance）的注册商标，该组织是一个同业公会，致力于推广无线 LAN 技术，同时提供互操作标准和测试。从技术角度说，某设备如果想印上 Wi-Fi 字样及标志，必须经过 Wi-Fi 联盟的认证。但实践中，Wi-Fi 可以用来指称任何基于 IEEE 802.11 标准的产品。

802.11 协议草拟于 1997 年，基本上是把当时的以太网标准（IEEE 802.3）照搬到了无线通信领域（基于以太网标准改写而成）。然而，直到 1999 年 802.11b 问世后，Wi-Fi 设备的市场才启动。由于这种技术相对简单，部署简便，而且工作于免许可的 2.4 GHz ISM 频段，任何人都可以利用它对既有的局域网进行“无线扩展”。今天，几乎所有台式机、笔记本、平板电脑和智能手机，乃至其他规格形制的设备都内置有 Wi-Fi 通信模块。

6.1　从以太网到无线局域网

802.11 无线标准主要是作为既有以太网标准（802.3）的扩展来设计的。事实上，以太网通常被称作局域网（LAN）标准，而 802.11 标准族（图 6-1）则相应地被称作无线局域网（WLAN，Wireless LAN ）标准。然而，熟悉历史的极客都知道，以太网协议很大程度上借鉴了 ALOHAnet 协议，而后者是第一个关于无线网络的协议，

由夏威夷大学开发并在 1971 年对外公布。换句话说，我们只是兜了个圈子而已！

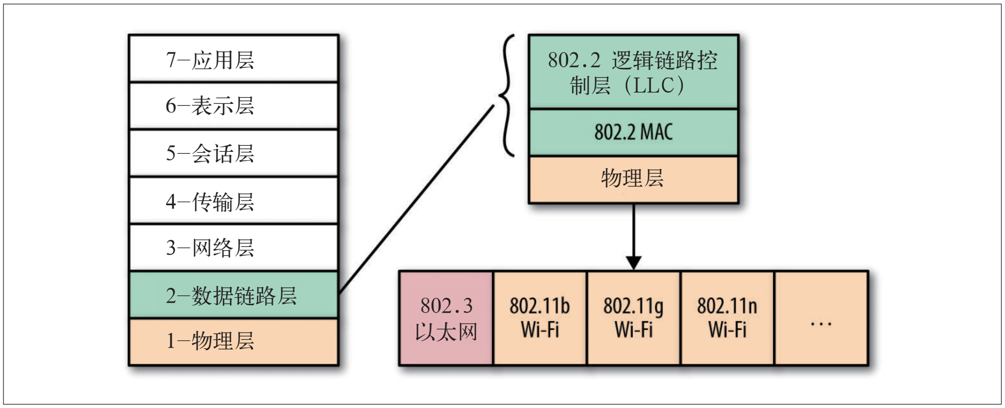


图 6-1：802.3（以太网）和 802.11（Wi-Fi）的数据和物理层

这个共同点非常重要，因为 ALOHAnet 协议影响了后来的以太网和 Wi-Fi 协议调度通信的方式。事实上，以太网和 Wi-Fi 都把共享媒体（无论是线缆还是无线电波）视为“随机访问通道”，即没有中心控制环节或者调度中心控制谁或哪台设备在哪个时刻可以发送数据。相反，所有设备都自我控制，大家必须协同工作，共同维护共享信道的性能。

以太网标准过去依赖于概率访问的 CSMA（Carrier Sense Multiple Access，载波监听多路访问）协议，这个名字听起来很复杂，实际上就是“先听后说”的一种算法。简单来说，如果你想发送数据，那么先要：

- 检查是否有人正在发送；
- 如果信道忙，监听并等待信道空闲；
- 信道空闲后，立即发送数据。

当然，任何信号传播都需要花时间，冲突也时有发生。为此，以太网标准也增加了冲突检测机制（CSMA/CD，Collision Detection）：如果检测到冲突，则双方都立即停止发送数据并小睡一段随机的时间（后续时间以指数级增长），从而保证发生冲突的发送端不会同步，且不会同时重新开始发送数据。

Wi-Fi 处理冲突的方式很类似，但也稍有不同：由于收发无线电的硬件所限，它不能在发送数据期间检测到冲突。实际上，Wi-Fi 采用的是冲突避免（CSMA/CA，Collision Avoidance）机制，即每个发送方都会在自己认为信道空闲时发送数据，以避免冲突。为此，每个 Wi-Fi 数据帧必须明确得到接收方的确认，以确保不发生冲突。

当然，这只是简单地概括而已。但这确实是以太网和 Wi-Fi 针对共享媒体调度通信的机制。对以太网来说，媒体就是物理线缆，对 Wi-Fi 而言，媒体则是无线信道。

实践中，概率访问模型在轻负载网络中表现很好。我们不会罗列数学证明，但要保证良好的信道利用率（冲突最小化），则信道负载必须低于 10%。保持低负载的情况下，不用太多协调和调度就能获得理想的吞吐能力。但是，随着负载增长，冲突次数也会迅速增加，造成整个网络性能不稳定。

> 高负载下性能不稳定的 Wi-Fi 网络很常见，比如某个技术大会现场，访问同一节点的客户端很多，这时候的 Wi-Fi 网络很可能就不稳定。当然，概率调度并不是造成这种结果的唯一因素，但无疑是其中之一。

6.2 Wi-Fi标准及功能

从 802.11b 开始，Wi-Fi 进入日常应用。但是，与任何流行的技术一样，IEEE 802 标准委员会并没有就此作罢，而是继续积极地发布新协议（表 6-1），以支持更高的吞吐量和更先进的调制技术、多流（multi-streaming）及其他各种新功能。

表6-1：Wi-Fi发布历史及路线图

802.11协议	发布时间	频率（GHz）	带宽（MHz）	流速率（Mbit/s）
b	1999-09	2.4	20	1、2、5.5、11
g	2003-06	2.4	20	6、9、12、18、24、36、48、54
n	2009-10	2.4	20	7.2、14.4、21.7、28.9、43.3、57.8、65、72.2
n	2009-10	5	40	15、30、45、60、90、120、135、150
ac	2014 年前后	5	20、40、80、160	最高 866.7

今天，b 和 g 标准已经得到非常普遍的支持。这两个标准使用的都是 2.4 GHz 的 ISM 频段、20 MHz 带宽，最多支持一个无线电数据流。根据当地管制，发射功率也很可能固定在最大 200 mW。有些路由器支持调整发射功率，但很可能会被区域上限值所覆盖。

那我们怎么提升未来 Wi-Fi 网络的性能呢？ n 和未来的 ac 标准将每个信道的带宽由 20 MHz 提升到 40 MHz，使用高阶调制算法，增加无线信道并行发射多个数据流（MIMO）。总起来说，在理想条件下，ac 无线标准的传输速度将达到 Gbit/s 以上。

6.3 测量和优化Wi-Fi性能

此时此刻，有读者可能会说，这只是“理想条件下”的理论速度啊！没错，Wi-Fi

网络的广泛部署和流行也导致了最大的一个性能挑战：小区内部和外部干扰。Wi-Fi 标准没有规定任何中央调度机制，因而对任何客户端的吞吐量和延迟都不提供保证。

新的 WMM（Wi-Fi Multimedia，Wi-Fi 多媒体）扩展会在无线电接口中对需要低延迟的应用（语音、视频等）启用基本的 QoS（Quality of Service，服务质量），但能识别的路由器很少，能识别的客户端就更少了。与此同时，你的网络内部和附近的 Wi-Fi 网络一定会争用共享的无线电资源。

或许你的路由器支持对网络中的客户端设置一些 QoS 策略（比如每个客户端或每种流量类型的最大速度），但对于附近其他 Wi-Fi 网络产生的流量，你则无计可施。Wi-Fi 网络之所以无处不在，主要原因就是它部署简单，而正因为部署简单，才会带来当前的性能问题。事实上，在城市繁华地段或者写字楼中，数十个 Wi-Fi 网络重叠的现象并不罕见（图 6-2）。

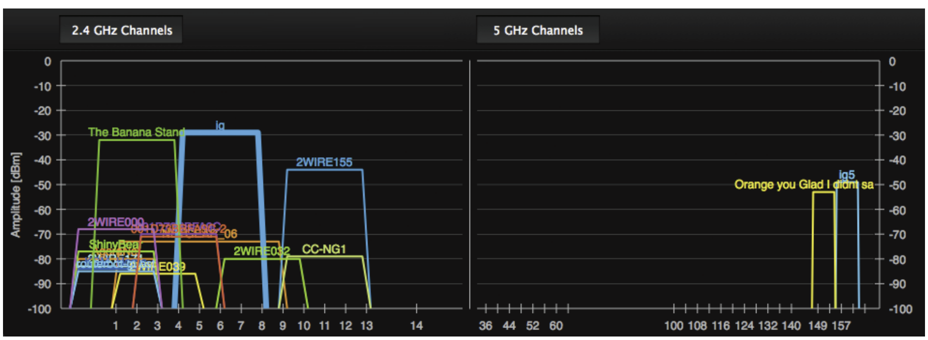


图 6-2：在 inSSIDer 中可视化地查看重叠的 Wi-Fi 网络（2.4 GHz 和 5 GHz 频段）

使用最多的 2.4 GHz 频段提供 3 个不重叠的 20 MHz 无线电信道：1、6 和 11（图 6-3）。当然，并不是所有国家都会这样分配。某些国家可能会允许使用更高的信道（13、14），而其他国家则可能会限制你只能使用更少的信道。但抛开地方管制不论，关键是要知道你的网络附近远远不只有两三个 Wi-Fi 网络，其中一些网络一定会重叠，从而共享相同频率范围内的带宽。

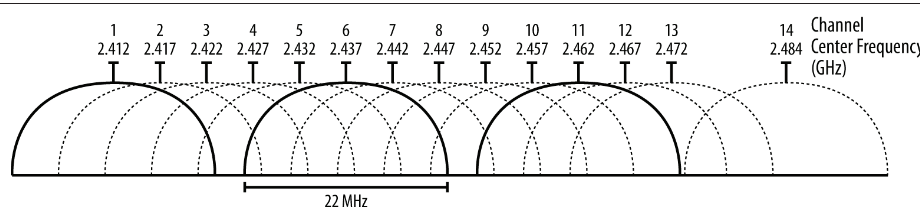


图 6-3：维基百科中关于 2.4 GHz 频段 Wi-Fi 信道的示意图

你的 802.11g 客户端和路由器的速度可能是 54 Mbit/s，但如果你的邻居也占用了相同的 Wi-Fi 信道，而且正在实时观看高清视频，那你的带宽就会减至一半，甚至还不到一半。你的接入点在这里没有发言权，因为这是 Wi-Fi 的功能，而非 bug！

更糟糕的是，延迟性能也好不到哪去。从客户端到 Wi-Fi 接入点的第一跳需要多长时间，也是没有保证的。在网络重叠严重的环境下，第一跳所需的时间可能是几十 ms，也可能是几百 ms。因为你需要跟同一信道内的其他无线客户端竞争！

好在，如果你积极地采用新技术，那很可能显著提升 Wi-Fi 网络的性能。新的 802.11n 和 802.11ac 标准使用 5 GHz 频段，不仅拓宽了频率范围，而且能保证在多数环境下不发生冲突。换句话说，至少在目前，如果你附近没有什么（像你一样的）技术大牛，那么一台双频路由器（支持 2.4 GHz 和 5 GHz）既能兼容 2.4 GHz 的老客户端，也能为支持 5 GHz 的客户端提供更好的性能。

测量 Wi-Fi 第一跳的延迟时间

运行 ping 命令是估计无线网络第一跳延迟时间的简便方式。但每次运行的结果可能不同，表 6-2 就是我在自己家里测试双频 802.11n 路由器得到的结果。

表6-2：2.4 GHz频段和5 GHz频段的时延差别

频率（GHz）	中值（ms）	95%（ms）	99%（ms）
2.4	6.22	34.87	58.91
5	0.90	1.58	7.89

拥挤的 2.4 GHz 频段与空闲得多的 5 GHz 频段（图 6-2）性能差异极大。2.4 GHz 频段中十几个重叠的网络导致第一跳的延迟长达 35 ms（95%），而我的笔记本电脑与路由器相距还不到 6 米。

综上所述，我们提到了关于 Wi-Fi 性能的哪些重要因素呢？

- Wi-Fi 不保证用户的带宽和延迟时间。
- Wi-Fi 的信噪比不同，带宽也随之不同。
- Wi-Fi 的发射功率被限制在 200 mW 以内。
- Wi-Fi 在 2.4 GHz 和较新的 5 GHz 频段中的频谱有限。
- Wi-Fi 信道分配决定了接入点信号会重叠。
- Wi-Fi 接入点与客户端争用同一个无线信道。

简单来说，根本不存在什么“典型的”Wi-Fi 性能。选用的标准、用户的位置、使用的设备，以及本地无线电环境决定了频率范围各不相同。如果你是唯一的 Wi-Fi 用户，那可以指望获得最大的吞吐量、最低的延迟，以及这两者的稳定状态。可如

果你是跟其他人共享一个接入点，而且近旁还有其他 Wi-Fi 网络，那可就没法说了，带宽和延迟都会高度不稳定。

Wi-Fi中的丢包

Wi-Fi 网络的设计天然会导致多个客户端的大量冲突。然而，即便如此，也不一定会导致更高的丢包率。所有 Wi-Fi 协议的数据和物理层实现都有自己的重发和纠错机制，这些机制向上层隐藏了重发操作。

换句话说，TCP 丢包在 Wi-Fi 网络中同样存在，因而其 TCP 层的传输速度不及大多数有线网络。除了直观的丢包问题，Wi-Fi 网络更突出的问题则是分组到达时间差异极大，这一切都要归咎于数据链路层和物理层的冲突及重发。

> 事实上，在 802.11n 之前，Wi-Fi 协议只允许同一时刻传输一个数据帧，这一帧必须得到链路层的确认才能继续发送下一帧。802.11n 则引入了新的"帧聚合"功能，从而支持同时发送和确认多个 Wi-Fi 数据帧。

6.4 针对Wi-Fi的优化建议

前面从整体上介绍了 Wi-Fi 网络突出的性能特点。实践中的 Wi-Fi 网络都还挺够用，而且其部署的简便性更是没法比拟。事实上，目前不支持 Wi-Fi 的计算机、手机或平板并不多见，而接入以太网的设备却常常需要外围设备支持。

有了这个总体认识，就可以讨论你的应用能否通过优化 Wi-Fi 网络受益了。

6.4.1 利用不计流量的带宽

现实当中的 Wi-Fi 网络一般都是有线局域网的扩展，而有线局域网可能又通过 DSL、有线电视网或者光纤连接到广域网。对于一般的美国人，平均上网带宽为 6.7 Mbit/s，而全球平均带宽是 2.6 Mbit/s（参见表 1-2）。换句话说，大多数 Wi-Fi 的速度最终仍受限于 WAN 的带宽，而不是其自身带宽。总之，无线网络通信风景这边独好！

不过，除了带宽瓶颈，这也意味着 Wi-Fi 网络通常要背靠不计流量的 WAN 连接——就算是有流量限制，其流量上限或吞吐量往往也会很高。因此，很多 3G 或 4G 用户由于成本和带宽原因不愿下载大文件，但 Wi-Fi 用户则不用担心这个问题。

当然，不计流量只是针对大多数情况而言的，如果 Wi-Fi 接入点后面是一个 3G 或 4G 连接，那得另当别论。基于这个事实，下载大文件、升级软件、流式播放等应用

都可以放心地通过 Wi-Fi 来做。换句话说，如果用户要做这些事，别忘了提醒他们切换到 Wi-Fi 网络上去！

> 很多移动运营商都推荐大数据应用“把流量转移到 Wi-Fi 上”，要么提醒用户切换到 Wi-Fi，要么在可能的情况下利用 Wi-Fi 在后台进行数据同步或大数据转移。

6.4.2 适应可变带宽

如前所述，Wi-Fi 不保证带宽和延迟时间。用户的路由器如果设置应用级别的 QoS 策略，那可能会为同一无线网络中的客户端提供一个公平的环境。然而，Wi-Fi 无线接口自身对 QoS 的支持非常有限。更糟糕的是，在多个重叠的 Wi-Fi 网络中根本没有 QoS 策略。

于是，每个客户端的实际带宽可能因位置、附近无线客户端的活动情况，以及无线大环境而每秒钟都会有所不同。

举个例子，高清视频流要求带宽为每秒达到 Mbit 级别（表 6-3），尽管大多数 Wi-Fi 标准在理想情况下都能满足这个要求，但实际应用中吞吐量间隙下降的情况并不少见。由于 Wi-Fi 网络的带宽天生具有不确定性，因此我们不能也不应该指望未来的下载比特率比过去高多少。在视频一开始下载时就测试带宽速率，可能会因回放期间无线条件的变化，而导致断断续续的缓冲暂停。

表6-3：采用H.264视频编解码方案的YouTube示例视频的比特率

容器	视频分辨率	编码	比特率（Mbit/s）
mp4	360p	H.264	**0.5**
mp4	480p	H.264	**1~1.5**
mp4	720p	H.264	**2~2.9**
mp4	1080p	H.264	**3~4.3**

虽然我们不能预测可用带宽，但我们可以，而且也应该利用连续的测量技术，比如自适应比特流，来主动适应带宽变化。

自适应比特流

自适应比特率并不适合所有资源，但对视频和音频这样的长时间流服务是非常合适的。

对视频而言，资源可能会使用多种比特率编码和存储，然后切割为多个部分（比如，YouTube 视频会分成多个 5~10 s 的块）。然后，在客户端下载视频流期间，客户端或服务器可以监控每个视频块的下载速度，必要时根据带宽的变化调整要下载的下一个视频块的比特率。事实上，现实中的视频服务，开始一般是低比特率的视频块，以便视频播放能更快开始。然后，再根据可用带宽的动态变化调整后续视频块的比特率。

每个资源要分别创建多少个比特率版本呢？取决于你的应用！不过，我可以告诉你，Netflix 为适应不同的屏幕大小和可用带宽，为每个视频流都创建了超过 120 个版本！让用户有流畅感、实时感，可真不是件简单的事儿。

6.4.3 适应可变的延迟时间

Wi-Fi 既不保证带宽，也不保证第一跳的延迟时间。如果要经过不止一跳（比如使用无线网桥作为接入点时），那预测延迟时间就更困难了。

理想情况下，如果没有干扰并且网络不满载，那么第一跳的延迟时间不会超过 1 ms，而且非常稳定。然而，现实当中，特别是高密度的城区或办公楼环境下，几十个 Wi-Fi 接入点和客户端会形成激烈的频率争用局面。

结果，第一跳中值 1~10 ms，后续中继的延迟时间 10~50 ms 都是合理的。在比较糟糕的情况下，高达几百毫秒也并非不可能。

如果你的应用最怕高延迟，那就要考虑在通过 Wi-Fi 运行时怎么调整其行为。事实上，有时候使用提供不可靠 UDP 传输的 WebRTC 倒不失为一个明智的选择。当然，传输方式的切换不能挽救无线网络，但却有助于降低协议和应用导致的延迟时间。

第 7 章

移动网络

据估计，到 2013 年上半年，全球移动上网设备大约为 64 亿台。IDC 市场分析报告称，仅 2012 年智能上网设备（智能手机、平板电脑、笔记本电脑、台式电脑等）的出货量就为 11 亿台。然而，更令人震惊的则是新设备出货量会呈直线增长：同一份 IDC 报告预测，到 2016 年，新设备出货量将达到 18 亿台。累计起来，到 2020 年全球移动上网设备将突破 200 亿台。

2012 年的全球人口约为 70 亿，2020 年将增至 75 亿。两个增长趋势一比，就可以看出人类对智能上网设备的需求是没有止境的。很明显，我们中的大多数人都不会满足于只拥有一部移动上网设备！

然而，上网设备的总数只是整个蓝图的一小部分。高速增长的出货量背后，是对同样没有止境的高速连接、无所不在的无线接入点，以及驱动这些新设备的在线服务的需求。因为如此，本章才会讨论 CDMA、GSM、HSPA 和 LTE 等各种蜂窝通信技术的性能问题。我相信，你的大多数用户都会使用这些技术中的一种访问你的站点或服务。赌注很高，前景很诱人，我们必须想办法赢下来。然而一提到性能，移动网络的一些设计方面的挑战就展现在了我们面前。

7.1　G字号移动网络简介

如果要全面解读各种蜂窝通信标准，它们的版本，以及每种技术的优劣势，那几章的篇幅肯定不够，至少要用几本书的篇幅。我们可没有那么雄心勃勃，在此我们只想就主宰未来无线技术的几种主要技术（表 7-1），讨论一下它们的运作特点，以及

这些特点带来的问题。

表7-1：四代移动网络

代	峰值数据速率	说明
1G	无数据	模拟系统
2G	Kbit/s	第一代数字系统，作为对模拟系统的替代或与之并存
3G	Mbit/s	专用数字网络，与模拟系统并行部署
4G	Gbit/s	数字及分组网络

首先要知道一个重要概念，即每一代无线技术都以其峰值频谱效率（bps/Hz）为标志，为了让用户更直观地理解，这个效率会转换成数据传输速率，比如 4G 网络的传输速率以 Gbit/s 来衡量。注意，前面提到了一个重要的关键词：峰值！提到这个词，大家可以回想一下 5.4 节“测量现实中的无线性能”中的介绍，所谓“峰值”指的是理想条件下测得的传输速率。

无论什么标准，每种网络真实的性能都会因提供商以及他们对网络的配置、每个小区内活跃用户的数量、特定位置的无线环境、使用的设备，以及影响无线性能的其他因素而异。明确了这个大前提，我们就可以合理地调整自己对性能的预期，比如对数据吞吐量可以按下界来估计，而对分组延迟时间则可以按上界来估计（表 7-2）。

表7-2：活动移动连接的数据速率和延迟时间

代	数据速率	延迟时间
2G	100~400 Kbit/s	300~1000 ms
3G	0.5~5 Mbit/s	100~500 ms
4G	1~50 Mbit/s	< 100 ms

当然，问题还会更复杂，因为所谓 3G 或 4G 的分类法其实很粗糙，相应地吞吐量及延迟时间也就更粗放了。为了搞清楚这里到底有什么事，以及这个产业的发展趋势，我们首先要了解一下这些技术的历史，包含技术发展背后的主要推动者。

7.1.1　最早提供数据服务的2G

最早的商业 1G 网络于 1979 年部署于日本。这种网络属于模拟系统，没有数据传输能力。1991 年，芬兰基于新兴的 GSM（Global System for Mobile communications，全球移动通信系统）标准建设了第一个 2G 网络，最早在无线电网络中引入了数字信令。这个网络支持基于电路交换的移动数据服务，比如短信（SMS），而且数据传输速率达到了惊人的 9.6 Kbit/s！

直到 1990 年代中期，GPRS（General Packet Radio Service，通用无线分组业务）被引入 GSM 标准，无线互联网才真正走向实用。当然，速度可能是很慢的：GPRS 传

输速度能达到 172 Kbit/s，往返时间通常为几百 ms。GPRS 与早期 2G 语音技术的结合通常被称为 2.5G。几年后，这些网络又加入了 EDGE（Enhanced Data rates for GSM Evolution，GSM 增强数据率演进）功能，将峰值速率提高到 384 Kbit/s。第一个 EDGE 网络（2.75G）是 2003 年在美国建成的。

此后，进入了一个停滞或者叫调整期。无线通信产业发展了几十年，而实用、面向消费者的移动网络数据服务还是最近才出现的！ 2.75G 网络几乎延用了 10 年，虽然是昨天的技术，但在世界各地仍然广泛使用。可今天，如果没有了高速无线网络，很难想象将是一番什么样的景象。无线技术的应用之广、发展之快令人咋舌。

7.1.2　3GPP与3GPP2

随着消费者对无线数据服务的需求开始增长，无线电网络互通变成了热门话题。对网络运营商而言，他们必须购买和部署无线接入网络（RAN）的硬件，而这需要巨额投资和持续维护的成本。显然，标准的硬件能降低成本。类似地，在缺乏行业标准的情况下，用户只能使用自己的家庭网络，这就限制了移动数据访问的用途和便捷性。

作为回应，ETSI（European Telecommunication Standards Institute，欧洲电信标准协会）在 1990 年代初期制定了 GSM 标准，该标准迅速被很多欧洲国家采用，并在全球推广。事实上，GSM 应该是部署最广泛的无线标准。据估计，GSM 占据了整个移动通信市场 80%~85% 的份额。但它并不是唯一的标准，与之同时，美国高通公司的 IS-95 标准也占据了 10%~15% 的市场份额，主要部署在北美地区如图 7-1 所示。于是，为 IS-95 无线网络设计的设备不能在 GSM 网络中使用，反之亦然。相信经常出国的读者一定对此深有感触。

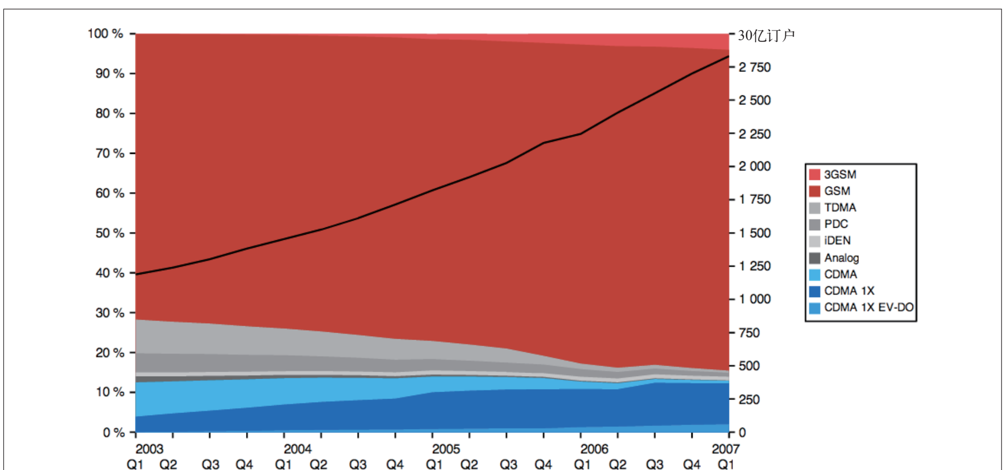


图 7-1：2003~2007 年移动标准市场份额（引自维基百科）

1998 年，在认识到既有标准需要全球演进之后，同时也为了制定下一代网络（3G）的标准，GSM 和 IS-95 组织又成立了两个全球性的合作伙伴项目。

- 3GPP（3rd Generation Partnership Project，第三代合作伙伴项目）
  负责制定 UMTS（Universal Mobile Telecommunication System，通用移动通信系统），这是 3G 技术与 GSM 技术结合的产物。这个项目后来也负责维护 GSM 标准和制定新的 LTE 标准。

- 3GPP2（3rd Generation Partnership Project 2）
  负责基于 CDMA2000，也就是高通制定的 IS-95 标准的后续技术制定 3G 规范。

结果，两种标准（表 7-3）及相关的网络基础设施并行推进。尽管步调可能不一致，但它们的底层技术大体都遵循了类似的发展路线。

表7-3：3GPP和3GPP2开发的蜂窝网络标准

代	组织	发布版
2G	3GPP	GSM
	3GPP2	IS-95（CDMA1）
2.5G、2.75G	3GPP	GPRS、EDGE（EGPRS）
	3GPP2	CDMA2000
3G	3GPP	UMTS
	3GPP2	CDMA2000 1x EV-DO Release 0
3.5G、3.75G、3.9G	3GPP	HSPA、HSPA+、LTE
	3GPP2	EV-DO Revision A、EV-DO Revision B、EV-DO Advanced
4G	3GPP	LTE-Advanced、HSPA+ Revision 11+

相信读者能在这里看到自己的熟悉的字眼儿：EV-DO、HSPA、LTE。很多网络运营商都投入了大量资源，并且还将一直投入下去，以推广这些他们所谓的“最新最快的移动数据网络”。 不过，我们关注这些历史问题可不是为了营销，而是为了从宏观上把握移动无线产业的演进：

- 世界上部署了两个主导的移动网络；
- 3GPP 和 3GPP2 共同管理每种技术的演进；
- 3GPP 和 3GPP2 标准不支持设备互操作。

不存在孤立的 4G 或 3G 技术。ITU（International Telecommunication Union，国际电信联盟）负责规划每一代无线网络的国际标准和性能特点，比如数据传输速率和延迟时间，而 3GPP 和 3GPP2 则负责制定相关标准，在各自相关技术的条件下满足甚至超过规划的预期。

怎么知道你的运营商在运行什么网络？很简单，你的手机有 SIM 卡吗？如果有，那就是源自 GSM 的 3GPP 技术。要了解有关网络的更多信息，可以再咨询运营商，或者如果手机支持，可以直接查看自己手机上的网络信息。Android 用户，打开手机，调出拨号盘，输入：*#*#4636#*#*。如果手机允许执行该命令，那应该看到一个诊断屏幕，从中可以看到移动连接的类型、电池诊断信息，等等。

7.1.3 3G技术的演进

3G 网络存在两个主导且互不兼容的标准：UMTS 和 CDMA，分别由 3GPP 和 3GPP2 制定。然而，就像早期的蜂窝标准一样，这两种标准各自都有自己的过渡性版本，通常被称为 3.5G、3.75G 和 3.9G（表 7-3）。

为什么不直接跳到 4G 呢？因为制定一个新标准需要很长时间，而且更重要的是，建设新网络需要巨额投资。稍后我们会介绍到，4G 的无线电接口及基础设备完全不同于 3G。为此，也为了很多购买了 3G 设备的用户的利益，3GPP 和 3GPP2 一直在既有的 3G 标准基础上改进，而这也能让运营商渐进地升级既有网络，为现有用户提供更好的性能。

毫无疑问，3G 网络每一种新标准的吞吐量、延迟时间及其他性能指标都有所提升，某些情况下甚至提升非常大。实际上，从技术角度讲，LTE 仍然属于 3.9G 的过渡性标准！然而，在介绍 LTE 之前，让我们先了解一下 3GPP 和 3GPP2 的各种里程碑性标准。

1. 3GPP技术的演进（表7-4）

表7-4：3GPP发布历史

版本	年份	说明
99	1999	UMTS 标准的第一个版本
4	2001	增加全 IP 核心网络
5	2002	增加 HSDPA（High Speed Downlink Packet Access）
6	2004	增加 HSUPA（High Speed Uplink Packet Access）
7	2007	增加 HSPA+（High Speed Packet Access Evolution）
8	2008	增加 LTE SAE（System Architecture Evolution）
9	2009	改进 SAE 与 WiMAX 的互操作性
10	2010	增加 4G LTE-Advanced 架构

在 3GPP 制定的标准中，HSDPA（High Speed Downlink Packet Access，高速分组下行接入）和 HSUPA（High Speed Uplink Packet Access，高速分组上行接入）通常被合称为 HSPA（High Speed Packet Access，高速分组接入）。这两个标准使实际的吞吐量达到了一位数的 Mbit/s 级别，在早期 3G 网络基础上有了明显提升。HSPA 网络经常被为 3.5G。

此后，下一个新版本是 HSPA+（3.75G），这个版本由于引入了简化的核心网络架构，降低了延迟时间，同时数据传输速率也实现了一位数 Mbit/s 级别吞吐量从中速到高速的提升。不过，HSPA+ 还不是终点，自发布之日起，该版本仍然不断改进，目前仍然在跟 LTE 和 LTE-Advanced 分庭抗礼！

2. 3GPP2技术的演进（表7-5）

表7-5：3GPP2 CDMA2000 1x EV-DO标准的发布历史

版本	年份	说明
Rel. 0	1999	1x EV-DO 标准的第一个版本
Rev. A	2001	提升峰值数据速率、降低延时、QoS
Rev. B	2004	在 Rev. A 基础上增加多运营商支持
Rev. C	2007	改进核心网络效率及性能

CDMA2000 EV-DO 标准是由 3GPP2 沿着类似的升级路径制定的。第一版（Rel. 0）把下行吞吐量提升到一位数 Mbit/s 级别。Rev. A 则提升了上行速率，更进一步，上下行速率在 Rev. B 中同时得到提升。实际上，Rev. B 网络的吞吐量已经达到了一位数 Mbit/s 级别的中高级水平，能够与 HSPA 及 HSPA+（即 3.5G 和 3.75G）网络媲美。

随后的 Rev. C 经常被称为 EV-DO Advanced，它在容量和性能方面有了很大提升。但是，EV-DO Advanced 的采标率并没有达到 HSPA+ 那样的水平。为什么？仔细看一看各代标准（表 7-3），就会发现 3GPP2 没有发布正式的、竞争性的 4G 标准！

虽然 3GPP2 可以继续改进其 CDMA 技术，但有一天，网络运营商和网络提供商一致认可 3GPP LTE 是各种网络共同的下一代 4G 标准。因此，很多 CDMA 网络运营商也率先投资建设了早期的 LTE 基础设施，以便在某种程度上保持对演进中的 HSPA+ 的竞争力。

换句话说，世界上的大多数移动运营商将把 HSPA+ 和 LTE 作为未来的移动无线标准。不用紧张，这应该是件好事。现有的 2G 和 3~3.75G 技术仍然是当前移动无线网络的主体，更重要的，它们至少还会继续为我们服务 10 年。

3G 经常被称为“手机宽带”。但是，宽带是一个相对的概念。有人认为宽带至少意味着 256 Kbit/s 的速率，也有人认为宽带速率应该在 640 Kbit/s 以上。而事实上，这个速率会因我们的上网应用而有所不同。随着服务水平的提高，以及人们对高速率需求的增长，宽带的速率也会进一步提高。考虑到这一点，不如把 3G 的速率想象成达到甚至超过 Mbit/s 级别。超过 Mbit/s 级别多少？那就要看相应的标准版本（如前所述）、运营商的网络配置，以及设备的能力了。

7.1.4 IMT-Advanced的4G要求

在介绍各种 4G 技术之前，有必要先了解一下“4G”这个标签背后有什么。与 3G 类似，也没有孤立的 4G 技术。所谓 4G，背后是一组具体要求（IMT-Advanced），这组要求是 ITU 在 2008 年就制定和公布了的。任何达到这些要求的技术，都可以看作是 4G 技术。

IMT-Advanced 的部分要求举例如下：

- 以 IP 分组交换网络为基础；
- 与之前的无线标准（3G 和 2G）兼容；
- 移动客户端的速率达到 100 Mbit/s，静止时的速率达到 Gbit/s 以上；
- 100 ms 控制面延迟，10 ms 用户面延迟；
- 资源在用户间动态分配和共享；
- 可变带宽分配，5~20 Mhz。

实际的要求比这些多得多，但这里的几个要求切中了我们这里讨论的主题：与前一代相比，拥有更高的吞吐量和更低的延迟时间。知道了这些条件，就该知道怎么划分 4G 网络了，对吧？不能这么说，这也太简单了！营销部门肯定会有不同的说法。

LTE-Advanced 作为一个标准，是专门为满足 IMT-Advanced 的所有条件而制定的。事实上，它也是 3GPP 为此而制定的第一个标准。不过，假如你前面认真阅读了，那应该会注意到 LTE（第 8 个版本）和 LTE-Advanced（第 10 个版本）是两个不同的标准。从技术角度讲，LTE 实际应该算是 3.9G 过渡性标准，尽管它的很多方面都达到了 4G 的要求，但还是不够全面！

不过，这正好是市场营销人员大显身手的地方。ITU 拥有 3G 和 4G 注册商标，因此其用途应该与每一代标准的要求吻合。但运营商在市场营销方面技高一筹，他们把“4G”商标包装成了包含一组几乎满足 4G 要求的技术。为此，LTE（第 8 个版本）和大多数 HSPA+ 网络虽然没有完全达到 4G 的要求，也都被作为 4G 技术来宣传炒作。

那到底有没有真正的 4G 技术呢？正在部署之中。但考虑到其前几代技术在市场上被炒出了别样的滋味，我们有必要提高警惕。无论如何，“4G”这个词儿如今在很多运营商那里都是含含糊糊的。要知道真相，你得认真看那些小字印刷的细则以理解这一技术。

7.1.5 长期演进（LTE）

尽管 3G 标准在不断演进，但随着对高速率和低延迟的需求越来越强烈，UMTS 技术设计上的内在局限性也暴露出来了。为弥补这些缺陷，3GPP 计划重新设计核心及无线网络，于是 LTE（Long Term Evolution，长期演进）标准应运而生。LTE 与前代标准的主要区别和特点如下：

- 核心网络全部为 IP 分组交换网；
- 简化了网络架构，降低建设成本；
- 用户面和控制面的低延迟时间（分别为 <10 ms 和 <100 ms）；
- 新无线接口及调制算法实现了高吞吐量（100 Mbit/s）；
- 可用于较大的带宽配置及运营商集群；
- 要求所有设备支持 MIMO。

毫不奇怪，这些特点与前面介绍的 IMT-Advanced 要求很类似。LTE（第 8 次发布）奠定了新网络架构的基础，而 LTE-Advanced（第 10 次发布）通过全方位的改进达到了 4G 的要求。

不过有一点必须说明，由于其无线及核心网络的实现的差异，LTE 网络可不是对已有 3G 基础设备的简单升级。相反，LTE 网络必须与现有 3G 基础设备并行部署，并采用不同的频率范围。不过，由于 LTE 是 UMTS 和 CDMA 的共同后继版本，因此它也提供了一种兼容二者的机制。由此，LTE 用户可以无缝切换到 3G 网络，并在 LTE 基础设施就绪后再迁移过来。

最后，顾名思义，LTE 的确是一个针对所有未来移动网络的长期演进规划。唯一的问题是这个未来有多远？有的运营商已经开始投资建设 LTE 基础设备，其他运营商也开始寻找频谱、资金。不过，根据业内人士估算，这次迁移肯定需要一个较长的周期，或许得需要十年左右。在此期间，HSPA+ 将扮演主角。

> 所有支持 LTE 的设备必须支持多射频，以满足 MIMO 的要求。可是，每个设备还需要单独的无线接口以兼容早期的 3G 和 2G 网络。这样算下来，每台设备至少要支持 3~4 个射频！为了实现 LTE 的高速率，甚至需要 4x MIMO，从而使射频总数达到 5~6 个。你是不是奇怪过自己的电池为什么那么不抗用？

7.1.6 HSPA+推进世界范围内的4G普及

HSPA+ 是 3GPP 在第 7 次发布时引入的，那是 2007 年。不过，尽管人们的眼球多被 3GPP 于 2008 年第 8 次发布的 LTE 所吸引，但不可忽视 HSPA+ 始终没有停止改进的步伐，仍然继续向前发展。事实上，HSPA+ 的第 10 次发布已经在很多方面达到了 IMT-Advanced 的标准！

不过，你可能会问，如果大家都认同 LTE 是未来移动网络的标准，那为什么还要继续改进和在 HSPA+ 上投入呢？答案很简单：成本。3GPP 的 3G 技术占据了全球无线市场的大部分份额，也就是说，世界各地的运营商在这方面已经投入了巨资。迁移到 LTE 必须建设全新的无线网络，而这又需要一大笔开支。相对而言，HSPA+ 的投入产出比更可取，运营商可以递增地升级现有网络，同时获得相对不错的性能！

投入产出是运营商的生命线，也是为什么业界预测（图 7-2）未来几年全球 HSPA+ 是向 4G 迁移的主体的原因。在此期间，3GPP2 的 CDMA 技术还将继续存在，但据预测其用户数量将缓慢减少，而新的 LTE 网络在不同地区将同步建设（速率可能不同）。这部分是因为成本，部分是因为管制和必要无线电频谱的可用性。

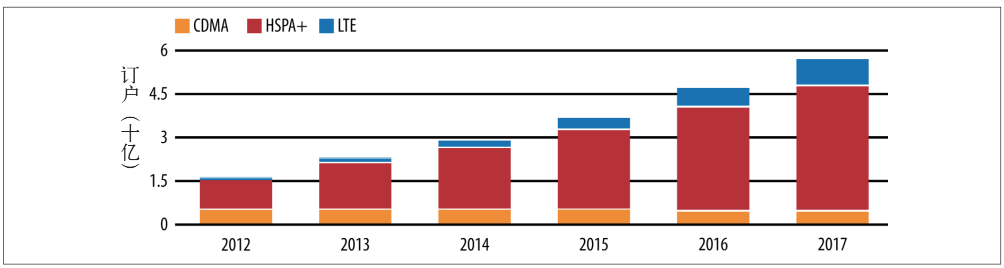


图 7-2：北美 4G：HSPA+ 和 LTE 移动宽带增长趋势

由于种种原因，北美地区的 LTE 建设似乎处于领先地位。目前的行业预测显示，到 2016 年，美国和加拿大的 LTE 用户将超过 HSPA 用户（图 7-3）。不过，北美的 LTE 部署速度明显比其他国家快很多。从全球来年，HSPA+ 仍然是接下十年中主导的移动无线技术。

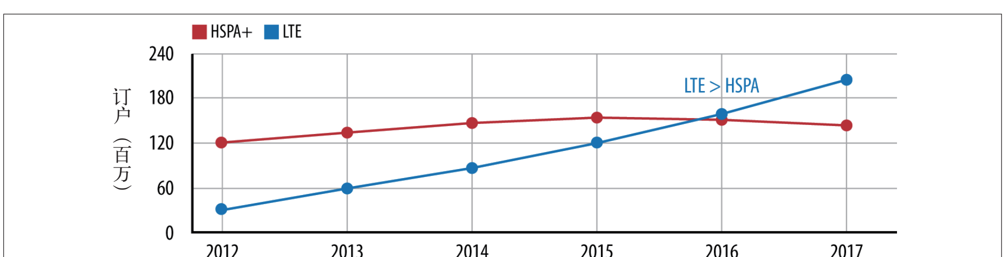


图 7-3：北美 4G：美国及加拿大 HSPA+ 和 LTE 增长趋势

虽然很多人看到 HSPA+ 与 LTE 趋势对比会感到惊讶，但这个结果其实并不意外。抛开其他因素不提，这个趋势至少能反映一个事实，即一个新的无线标准要在现实中成为主流技术，大概要花 10 年时间。

放眼未来，应该说到 2020 年代早期 LTE-Advanced 成为主流不是梦！遗憾的是，新无线基础设施的建设总是那么烧钱、费时。

7.1.7 为多代并存的未来规划

通信行业是容不得半点主观臆断的。但到目前为止，我们已经介绍了足够的背景，应该能够就未来的移动网络和先进程度作出合理的预期。

首先，无线标准发展很快，但这些网络的物理设施建设则既要花钱又得花时间。将来，如果这些网络部署完成，那势必还要投入很多时间维护以收回成本，保证服务品质。换句话说，尽管市场上对 4G 的宣传炒作沸沸扬扬，老一代网络至少还得服务社会十年以上。所以，在创建移动 Web 应用时，应该考虑这一点。

让人觉得讽刺的是，虽然 4G 网络在 IP 数据传输方面的性能有了很大提升，但 3G 网络在处理老式语音通信方面反而更有效率！ VoLTE（Voice over LTE）正在积极制定，目标是通过 4G 网络高效可靠地传输语音。但目前，大多数 4G 网络仍然要靠原来的电路交换设备传输语音。

自然地，在面向移动网络构建应用时，不能只考虑一种网络，更不能寄希望于特定的吞吐量或延迟时间。前面已经介绍过，任何网络的实际性能都具有高度可变性，取决于部署的版本、基础设施、无线条件，以及其他众多的因素。我们的应用要能适应不断变化的条件，吞吐量、延迟时间，甚至无线连接的有无，都可能变化。如果用户正移动着，那么很可能要跨越几代不同的网络（LTE、HSPA+、HSPA、EV-DO，甚至 GPRS Edge），具体就要看信号强度和覆盖范围了。如果你的应用没有考虑这一点，用户体验就会下降。

好在，HSPA+ 和 LTE 的采用速度很快，这样那些依赖高吞吐量和低延迟时间的应用就可以脱颖而出。这两种新标准的吞吐量和延迟时间指标是很惊人的（表 7-6）：现实中的吞吐量能达到一位数 Mbit/s 的中高级别，延迟时间则低于 100 ms。这两个指标已经堪比很多家庭和办公室的 Wi-Fi 网络了。

不过，尽管 4G 无线性能经常可以媲美 Wi-Fi，甚至有线网络，但将它们等同视之则是不应该的，不，应该说是绝对不可以的。

表7-6：HSPA+、LTE及LTE-Advanced的对比

	HSPA+	LTE	LTE-Advanced
下载峰值（Mbit/s）	168	300	3000
上传峰值（Mbit/s）	22	75	1500
最大 MIMO 流	2	4	8
空闲到连接的延迟（ms）	< 100	< 100	< 50
休眠到活动的延迟（ms）	< 50	< 50	< 10
用户面单向延迟（ms）	< 10	< 5	< 5

比如说，大多数用户和开发人员都期待“一直在线”的体验，即设备始终连接着互联网，随时可以响应用户输入或到来的数据。这种期待在无线网络环境没有什么问题，但对无线网络而言绝对是不现实的。电池容量和设备能力等现实因素，让我们不得不明确考虑移动网络的局限性。为了加深理解，让我们再进一步。

用户面单向延迟

用户面单向延迟是 LTE 标准规定的一个目标时间，指的是一个分组在无线设备与无线发射塔之间单向传输的时间。换句话说，就是设备处于大功率连续接收状态时，第一跳的延迟时间。任何应用的数据传输都必须付出这个代价，没有捷径。

7.2 设备特性及能力

经常被人忘记的一个事实，就是现有无线网络只是问题的一半。另一半当然是来自不同厂商的设备，以及它们的上市时间，这些设备各有各的特点。比如，CPU 速度和核心数量、内存大小、存储能力、有无 GPU 等。这些因素中的任何一个都会影响设备以及运行于其上的应用的整体性能。

不过，即使考虑到了上述所有因素，对于网络性能而言，还是有一个经常被忽视的方面：无线电收发能力。特别地，用户手里拿着的设备必须能利用无线基础设施！运营商可能会部署最新的 LTE 网络，但上市较早的设备可能根本就不支持该网络，反之亦然。

用户设备分类

3GPP 和 3GPP2 标准都在持续演进，不断提高对无线接口的要求，包括调制方法、射频数量，等等。为了获得最佳性能，设备必须达到每种网络对特定 UE（User Equipment，用户设备）类别的要求。事实上，每次发布标准的新版本时，经常会有多个 UE 类别，每个类别分别对应着不同的无线性能。

明显但又十分重要的一个问题就是，为什么？跟往常一样，答案很简单：成本。多个设备类别就是认可设备的差异化，它们的价格可以不同，以适应不同的收入水平的用户，它们的能力可以不同，以适应当前部署的不同的网络。

单单 HSPA 标准就规定了 36 个可能的 UE 类别！事实上，你说自己有一个“支持 HSPA 的设备”（表 7-7），这句话还是不够确切。比如，假设无线网络可用，那要获得 42.2 Mbit/s 的吞吐量，必须有一个类别 20（2x MIMO）或类别 24（双小区）的设备。更麻烦的是，类别 21 的设备（23.4 Mbit/s 的峰值速率）并不保证会比类别 20 的设备具有更高的吞吐量。

表7-7：3GPP HSPA UE类别示例

3GPP版本	类别	MIMO，多小区	峰值速率（Mbit/s）
5	8	—	7.2
5	10	—	14.0
7	14	—	21.1
8	20	2x MIMO	42.2
8	21	双小区	23.4
8	24	双小区	42.2
10	32	四小区 + MIMO	168.8

类似地，LTE 标准也定义了自己的 UE 类别（表 7-8）：高端的智能手机一般是类别 3~5 的设备，但它也有可能与类别 1~2 中更廉价的设备共享网络。UE 类别编号越大，比如要求 4x 或 8x MIMO，越有可能是专用设备。但多路无线信号同时发射也会消耗更多电量，对一般用户来讲恐怕并不实用。

表7-8：LTE UE类别

3GPP版本	类别	MIMO	峰值下载速率（Mbit/s）	峰值上传速率（Mbit/s）
8	1	1x	10.3	5.2
8	2	2x	51.0	25.5
8	3	2x	102.0	51.0
8	4	2x	150.8	51.0
8	5	4x	299.6	75.4
10	6	2x 或 4x	301.5	51.0
10	7	2x 或 4x	301.5	102.0
10	8	8x	2998.6	1497.8

现实中，大多数早期 LTE 网络都面向类别 1~3 的设备，而早期 LTE-Advanced 网络则主要将类别 3 作为主要的 UE 类别。

如果你有一台 LTE 或 HSPA+ 设备，那你知道它属于哪个类别吗？知道了所属类别之后，那你知道网络运营商正在运行的是 3GPP 发布的哪个版本吗？要想获得最佳性能，这两方面必须匹配。否则，无线网络或手机中的一方就会成为瓶颈。

移动设备的无线电规范解密

如果你看过自己移动设备的技术规格，那“无线与蜂窝网络”部分罗列的大量频率和技术一定会让你头晕眼花。不过，有了之前的介绍，解密这些频率和技术应该就不难了。举个例子吧，我们看一看谷歌 Nexus 4 的技术规格：

- GSM/EDGE/GPRS（850、900、1800、1900 MHz）
- 3G（850、900、1700、1900、2100 MHz）
- HSPA+ 42

第一行告诉我们这台设备可以在 2G 网络中运行，而且支持 GPRS（2.5G）和 EDGE（2.75G），峰值速率可达几百 Kbit/s。列出的频率表示可以收发的无线电频谱，以适应不同地区的频谱管制及网络部署情况。

第二行也类似，只不过没有标示出最大的 3G 吞吐量。不过，第三行揭示了这个信息：HSPA+ 表示这部手机可以在 3.75G 网络中运行，数字“42”的意思是该设备要么属于支持 MIMO 的类别 20，要么属于支持双小区的类别 24，最高下行速率可达 42.2 Mbit/s（当然是在网络允许的条件下）。事实上，Nexus 4 是一部类别 24 的双小区设备。

最后，我们知道这部手机不支持 LTE，要支持 LTE 就需要除 2G、3G 之外的另一套无线接口。在很多手机都已经做得非常小巧的条件下，居然还能同时内置多套无线电收发装置（大多数内置有 2~4 套），简直叹为观止！

7.3　无线电资源控制器（RRC）

3G 和 4G 网络都有一个独特的装置，这个装置在有线网甚至 Wi-Fi 中都是不存在的。这个装置就是无线电资源控制器（RRC，Radio Resource Controller ）。RRC 负责调度协调移动设备与无线电基站之间所有的通信连接（图 7-4）。理解为什么要有这么一个装置，以及它对移动网络的性能有什么影响关系到能否构建高性能的移动应用。RRC 直接影响延迟、吞吐量和设备电池的使用时间。

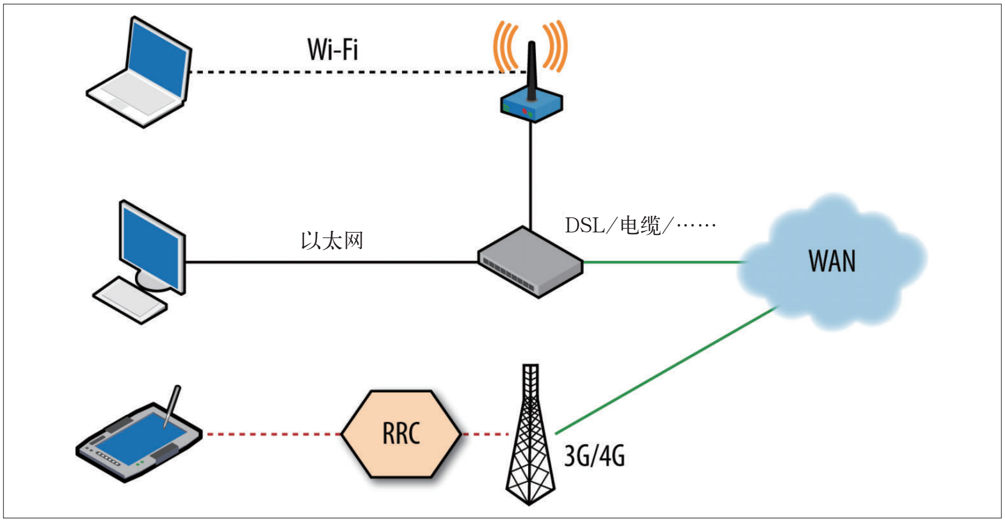


图 7-4：无线电资源控制器

使用物理连接方式，比如以太网线连接时，计算机直接连接网络并且始终在线，因此连接两端可以随时发送数据。在这种情况下，延迟时间才可能降到最短。就如 6.1 节“从以太网到无线局域网”所述，Wi-Fi 标准的连接方式也是类似的，即联网的设备之间可以随时通信。这种情况下，延迟时间也有可能降到最低，但由于设备共享无线电介质，如果活动用户很多，那么也会导致冲突和较大的性能变化。另外，由于 Wi-Fi 网络中的任何一端随时可以发送数据，那其他设备必须随时做好接收数据的准备，因此无线电收发必须始终开启，从而消耗很大电量。

实践中，保持 Wi-Fi 无线信号不间断代价太大，因为大多数设备的电池容量都有限。为此，Wi-Fi 标准也有一个优化策略，即无线接入点在向某个客户端发送数据之前，会先通过一个周期性的信号帧广播 DTIM（Delivery Traffic Indication Message，发送数据指示消息）。相应地，客户端会收听这些 DTIM 帧，以便做好接收数据的准备。其他时间，无线电装置可以处于休眠状态。这个优化策略省了电，却增加了延迟时间。

> 未来的 WMM（Wi-Fi Multimedia，Wi-Fi 多媒体）标准会进一步提升 Wi-Fi 设备的电源使用效率，主要是会采用新的 PowerSave 机制，比如 NoAck 和 APSD（Automatic Power Save Delivery，自动省电传输）。

3G 和 4G 网络也存在同样的问题：平衡网络传输效率与电源使用效率。换句话说，移动设备受限于电池容量，经常会电源告急，而一个小区内的活跃用户又都希望网络效率较高，这就产生了矛盾。于是，RRC 出现了。

顾名思义，无线电资源控制器全权负责谁在什么时候能发言，带宽多大，功率多少，以及监控每个设备的电源状态等。简而言之，RRC 就是无线电网络的大脑。想要通过无线信道发送数据？你必须先向 RRC 申请无线电资源。想要接收互联网上的数据？ RRC 会通知你什么时候监听接收到来的分组。

好消息是，RRC 管理全都由网络负责。坏消息是，你不能通过 API 适当地控制 RRC，假如想针对 3G 或 4G 网络优化自己的应用，那就得知道并在 RRC 的限制之内操作。

> RRC 存在于无线电网络中。2G 和 3G 网络中的 RRC 处于核心运营网络，而在 4G 中，为提升性能、减少协调时间，RRC 被转移到了无线信号塔（eNodeB）。

7.3.1　3G、4G和Wi-Fi对电源的要求

无线电在任何手持设备中都是最耗电的组件之一。要说最耗电，当然是点亮的屏幕了，注意是点亮时的屏幕。但在实际使用中，屏幕一般情况下都是熄灭的，反倒是无线电组件，为了让用户有“永远在线”的体验，必须一直运行。

实现这个目标的一种方式就是让无线电组件不停地工作，可即使是当今最大容量的电池，这么干也会在几小时内把电量耗尽。况且，3G 和 4G 的最新版本要求并行传输（MIMO、多小区，等等），这相当于同时开启了多个无线电组件。实践中，必须在保持无线电开启以降低通信延迟，与保持低电量消耗以延长待机时间之间取得平衡。

不同的技术有什么不同，哪一种更有利于延长电池寿命？答案不一而足。在 Wi-Fi 网络中，每个设备自己设定传输功率，通常是 30~200 mW。不同的是，3G/4G 无线发射功率是由网络说了算，空闲状态下最低为 15 mW。然而，考虑到要覆盖更大范围以及排除干扰，这两个网络在通信时要求的最高功率会达到 1000~3500 mW！

实践中，在传输大量数据时，如果信号强度足够大，Wi-Fi 通常效率更高。不过，如果设备经常空闲，则 3G/4G 网络的效率更高。要获得最佳性能，最理想的做法就是能够在不同类型的连接之间动态切换。然而，至少目前来看，还没有这么一种机制。但这个领域的研究十分活跃，产业界和学术界都非常积极。

好了，电池和功率管理到底怎么影响性能呢？“信号功率”是达成高吞吐量的一个主要手段。然而，信号功率越高，消耗的能量也会显著增多，因而是可以对其加以节流，以期延长电池寿命的。同样，停止无线电信号发射，也可能会断开设备与发射塔之间的数据信道。而这就意味着每次数据传输之前，都必须经过一番控制信息

的交换，从而导致几十甚至几百 ms 的网络延迟。

吞吐量和延迟时间都与设备的功率管理策略直接相关。事实上，3G 和 4G 网络中的无线电功率都是由 RRC 控制的（这是重点），RRC 不仅会告诉你什么时候开始通信，还会告诉你发射功率多大，以及什么时候切换到另一种功率。

7.3.2 LTE RRC状态机

所有 LTE 设备的无线电状态都由当前为用户提供服务的无线信号塔控制。事实上，3GPP 标准定义了一个完备的状态机，这个状态机描述了连接到网络的每个设备的功率状态（图 7-5）。网络运营商可以修改相应的参数，以触发状态切换，而状态机自身则对所有 LTE 基础设施和设备一视同仁。

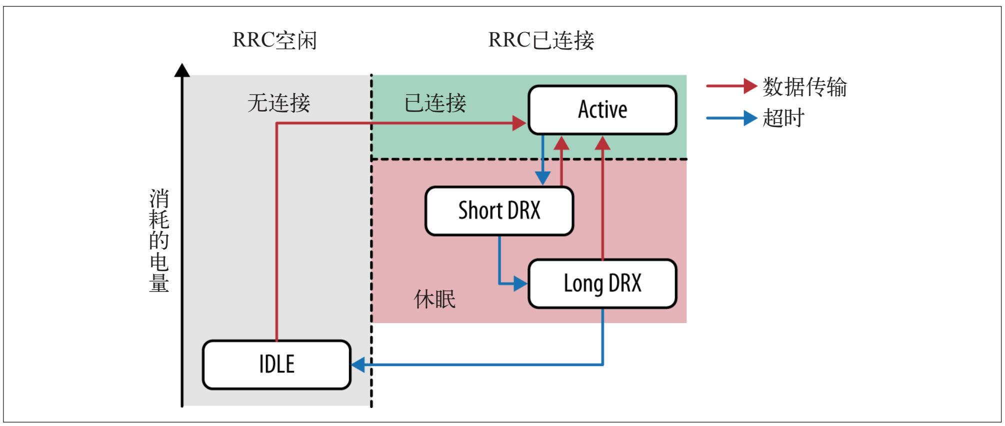


图 7-5：LTE RRC 状态机

- RRC空闲

  设备的无线电模块处于低功率状态（<15 mW），只监听来自网络的控制信号。运营商网络中的客户端没有无线电资源。

- RRC连接

  设备的无线电模块处于高功率状态（1000~3500 mW），要么传输数据，要么等待数据。运营商网络中指定了数据承载方式，也分配了专用的无线电资源。

简单来说，设备要么处于空闲状态，要么处于连接状态。处于空闲状态时，设备只监听控制信道的广播，比如准备接收数据的通知；处于连接状态时，网络通信环境就绪，相应资源也将分配给客户端。

处于空闲状态时，设备不能发送或接收任何数据。要收发数据，设备必须先通过监

听网络让自己与网络同步，然后再向 RRC 发送一个将其切换到“已连接”状态的请求。这个协商过程可能需要几次往返，3GPP LTE 规范对这个状态切换的容许条件是小于等于 100 ms。在 LTE-Advanced 中，这个切换时间进一步缩短为 50 ms。

切换到连接状态后，无线信号塔与 LTE 设备之间的网络环境准备就绪，随时可以传输数据。然而，如果通信的一方结束了数据传输，那 RRC 怎么知道什么时候把设备切换到低功率状态呢？这个问题问到点子上了，答案是不会！

IP 通信具有突发性，优化的 TCP 连接是持久的，而 UDP 通信就没有设计提供“传输结束”指示。结果，与 3.2.1 节介绍的 NAT 不同，RRC 状态机依赖于一组计时器来触发 RRC 的状态切换。

最后，由于连接状态需要的功率很高，为了更有效地完成操作，就有了多个子状态（图 7-5）。

- 连续接收
  最高功率状态，网络环境就绪，已分配网络资源。

- 短不连续接收（短DRX）
  网络环境就绪，未分配网络资源。

- 长不连续接收（长DRX）
  网络环境就绪，未分配网络资源。

在高功率状态下，RRC 为设备保留信道以接收和发送数据，然后将时隙、发射功率、调制模式等十余个参数通知设备。此时，如果设备空闲时间已经达到了配置的时间，则切换到短 DRX 状态，即网络环境仍然保持，但不分配特定的无线电资源。在短 DRX 状态下，设备只会监听网络周期性的广播，从而节省电能——与 Wi-Fi 的 DTIM 间歇不同。

什么是“分配的无线电资源”

与大多数现代无线标准一样，LTE 也共享上行和下行无线电信道，对这个信道的访问权由 RRC 控制。在连接状态下，RRC 告诉所有设备哪个时隙分配给了谁，必须使用多大的发射功率、调制方式，以及其他十余个参数。

如果移动设备没有得到 RRC 分配的这些资源，那它就不能发送和接收任何用户数据。因此，在处于 DRX 状态时，设备与 RRC 同步，但还没有被分配上行或下行资源。此时的设备处于“半醒”状态。

如果无线电空闲了足够长的时间，则设备切换到“长 DRX”状态。除了在被唤醒以监听广播之前要休眠更长时间之外，长 DRX 与短 DRX 状态是一样的（图 7-6）。

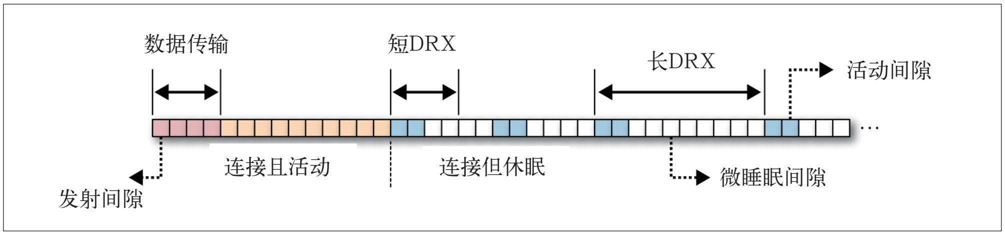


图 7-6：不连续接收：短 DRX 与长 DRX

如果网络或移动设备必须在无线电处于短 DRX 或长 DRX（休眠）状态下传输数据，怎么办？设备和 RRC 必须先交换控制信息，以协商何时开始传输，何时监听无线电广播。对 LTE 而言，协商时间（从休眠到连接）规定为 50 ms 以内，而在 LTE-Advanced 中，这个时间将被进一步缩短为 10 ms。

这些在实际当中都意味着什么呢？取决于实际的功率，LTE 设备可能要先花 10 到 100 ms 与 RRC 协商必要的资源（表 7-9），然后才能通过无线连接传输应用数据，通过运营商网络，再到公共互联网。在设计应用特别是对延迟敏感的应用时，是否考虑到了这些延迟，决定了你的应用是真正经过了优化，还是“性能不好”。

表7-9：LTE与LTE-Advanced中的RRC延迟

	LTE	LTE-Advanced
空闲到连接的延迟	< 100 ms	< 50 ms
DRX 到连接的延迟	<50 ms	< 10 ms
用户面单向延迟	< 5 ms	< 5 ms

7.3.3 HSPA与HSPA+（UMTS）RRC状态机

LTE 及 LTE-Advanced 之前的 3GPP 网络具有类似的 RRC 状态机，而且同样由无线电网络维持。这是我们喜闻乐见的一方面。但另一方面，早期网络的状态机还要更复杂一些（图 7-7），延迟时间长很多。事实上，之所以 LTE 的性能更好，就是因为它简化了状态机的架构，从而提升了状态切换的性能。

- 空闲

  与 LTE 类似。设备的无线电模块处于低功率状态，只监听来自网络的控制信号。运营商网络中的客户端没有无线电资源。

- Cell DCH

  连续接收时与连接状态的 LTE 类似。设备的无线电模块处于高功率状态，为上下行数据流的传输分配网络资源。

- Cell FACH

  设备无线电模块处于中等功率状态，比 DCH 状态消耗的电量少很多。设备没有专用的网络资源，但能通过共享的低速信道（通常不足 20 Kbit/s）传输少量用户数据。

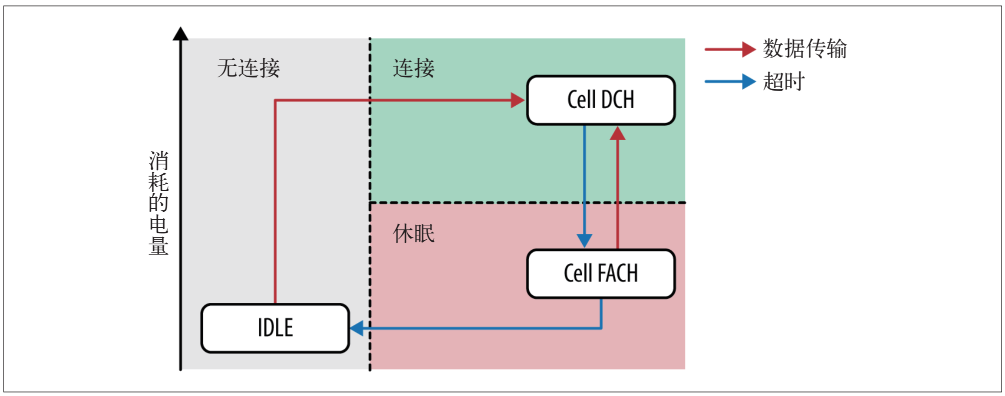


图 7-7：UMTS RRC 状态机：HSPA、HSPA+

空闲和 DCH 状态与 LTE 中的空闲和连接状态基本相同。但是，中间的 FACH 状态则是 UMTS 网络（HSPA、HSPA+）所特有的，这个状态下可以通过公用信道传输少量数据——慢、稳定，消耗的电量只有 DCH 状态的一半。实践中，这种状态多用于处理非交互通信，比如很多后台应用的轮询和状态检测。

毫不意外，DCH 到 FACH 状态的切换是通过计时器实现的。可是，FACH 到 DCH 的触发器是什么？每部设备都会缓存一些要发送的数据，只要数据量不超过网络配置的阈值（通常是 100~1000 字节），设备就会一直处于中间状态。而且，如果处于 FACH 状态一定时间后，仍然没有要传输的数据，则另一个计时器就会把设备切换到空闲状态。

> 与提供了两个中间状态（短 DRX 和长 DRX）的 LTE 不同，UMTS 设备只有一个中间状态：FACH。然而，即便从理论上说，LTE 能够做到对电量的更优控制，但无线电模块本身还是会消耗 LTE 设备的电量。毕竟，更高的吞吐量需要更多的电量作支撑。实际上。LTE 设备的耗电量比之前的 3G 设备高得多。

除了不同的功率状态之外，早期 3G 网络与 LTE 的最大区别恐怕就是状态切换的延迟了。LTE 网络从空闲到连接状态的延迟只有几百 ms，同样从空闲到 DCH 状态的切换在 3G 网络中则需要多达两秒，而且 3G 设备与 RRC 之间还要交换几十次控制消息！ FACH 到 DCH 的切换也好不到哪去，最长达一秒半。

好在，最新的 HSPA+ 网络对此做出了重大改进，已经可以同 LTE 媲美了（表 7-6）。不过，我们不能指望 4G 或 HSPA+ 一统天下，很多旧 3G 网络仍然要继续存在至少 10 年。所以，任何移动应用在通过 3G 接口访问网络时，都要考虑由 RRC 导致的长达几秒钟的延迟。

7.3.4 EV-DO（CDMA） RRC状态机

尽管 HSPA、HSPA+ 和 LTE 等 3GPP 标准是目前全球主流的网络标准，但我们也不能忽略 3GPP2 CDMA 网络的存在。EV-DO 网络的发展虽然比较缓慢，但据行业预测，2017 年的 CDMA 用户也将达到 5 亿！

不用说，虽然标准之间存在差别，但 UMTS 和 CDMA 网络的局限性却是共同的：电池电量是约束条件，而无线电模块非常耗电；网络效率都需要提高。因此，CDMA 网络同样具有 RRC 状态机（图 7-8），用于控制每部设备的无线电模块的状态。

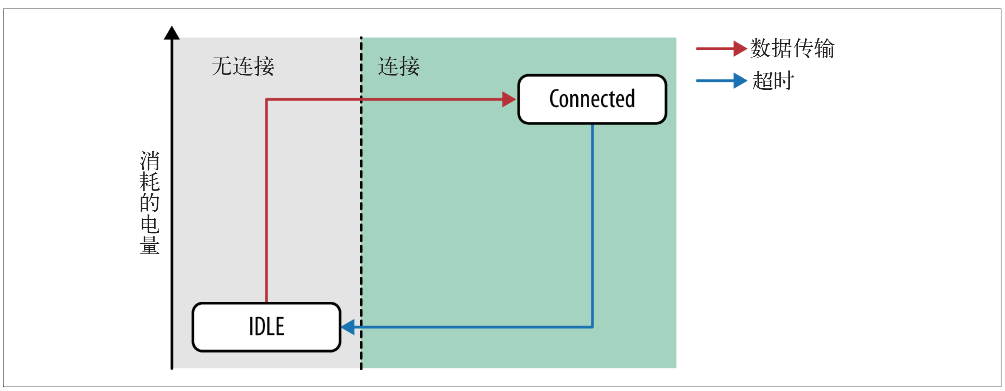


图 7-8：CDMA RRC 状态机：EV-DO（Rev. 0 - DO Advanced）

- 空闲

  与 3GPP 标准类似。设备的无线电模块处于低功率状态，只监听来自网络的控制信号。运营商网络中的客户端没有无线电资源。

- 连接

  与 LTE 的连接和 HSPA 中的 DCH 状态类似。设备的无线电模块处于高功率状态，为上下行数据流的传输分配网络资源。

这是我们见过的最简单的状态机：设备要么处于高功率状态，获得网络资源，要么空闲。任何网络传输都要切换到连接状态，相应的延迟时间与 HSPA 网络差不多，大概有几百或几千 ms（视基础设施的先进程度而不同）。没有中间状态，而且向空闲状态过渡也由运营商配置的超时时间控制。

7.3.5 低效率的周期性传输

不管是哪一代或底层使用了什么标准，由超时时间控制的无线电状态切换都会带来严重的后果：你的应用在访问网络时很容易既耗电，体验又不顺畅。因为必须等待足够长的时间，才能让无线电模块切换到低功率状态，然后再通过网络访问触发 RRC 状态切换！

为说明这个问题，假设有一部使用 HSPA+ 网络的设备，该设备被配置为在无线电模块停止工作 10 s 后从 DCH 切换到 FACH 状态。然后，我们加载一个应用或页面，周期性地传输数据，比如要进行实时查询，每 10 s 一次。结果如何呢？设备可能会花几百 ms 传输数据，之后的高功率状态完全浪费！更糟糕的是，由于传输是间隙性的，设备永远没机会切换到低功率状态，从而导致电池很快耗尽。

简言之，每一次无线电传输，不管数据量有多么少，都会切换到高功率状态。传输完成，必须等待计时器超时才能切换回低功率状态（图 7-9）。传输的数据量大小不影响计时器的超时时长。而且，设备在切换到空闲状态前，可能必须先经过几个中间状态。

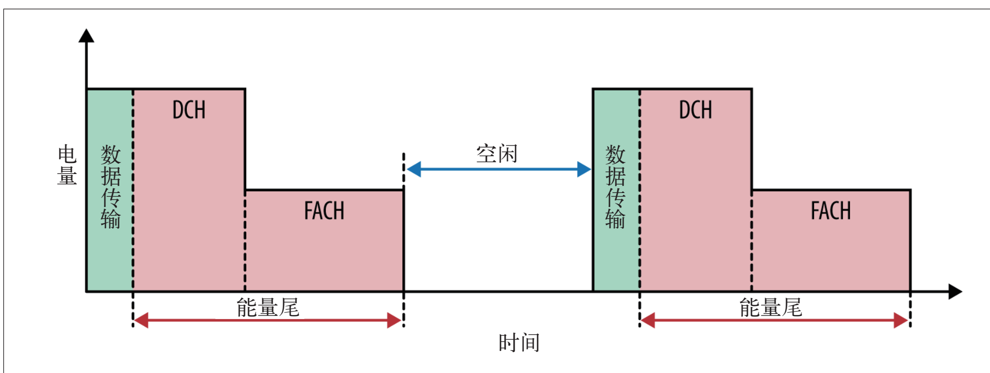


图 7-9：HSPA+ 存在能量尾，因为存在 DCH > FACH > IDLE 切换

计时器控制的状态切换导致“能量尾”（energy tail），“能量尾”导致移动设备在网络访问中低效的周期性传输。首先，状态切换有一个延迟时间，然后，传输开始，最终无线电模块空闲，白白地耗电，直到所有计时器超时，设备才切换到低功率状态。

46% 的电量消耗仅传输 0.2% 的数据

AT&T Labs Research 发表过一篇关于移动应用资源占用的研究论文（“Profiling Resource Usage for Mobile Applications”），分析了很多流行移动应用的网络和电池效率。其中，Pandora 是无线网络中低效率间隙性网络访问的代表。

Pandora 用户在播放歌曲时，应用会把整首歌曲以流的形式一次性下载下来。这样做是正确的，占有尽量多的数据，然后尽可能早一点关闭无线电模块嘛。可是，在下载歌曲之后，该应用会启动一个周期性的听众评测，每隔 60 秒就发送一次分析数据。最终结果如何？分析信标数据只占总传输字节的 0.2%，但消耗的电量却占应用耗电总量的 46%！

信标数据很小，但 RRC 状态切换导致的能量尾让无线电模块长时间处于高功率状态，从而浪费了 46% 的电量。事实上，把这些分析数据聚合为更少的请求，或者在无线电模块处于活动状态时再发送这些听众数据，可以消除不必要的能量尾，从而将应用的耗电量减少一半！

7.4 端到端的运营商架构

了解了 RRC 和设备之后，接下来我们把视角放得更大一些，看一看运营商网络中端到端的架构。我们的目标并不是成为这方面的专家，只是想重点介绍那些直接影响运营商网络中数据流的因素，以及它们影响我们应用性能的原因。

运营商网络中的基础设施，以及各种逻辑和物理组件的名字，取决于部署的网络属于哪一代和什么类型：EV-DO 或 HSPA、3GPP 或 3GPP2，等等。然而，这中间还是有很多类似的地方，本节就来介绍一下 LTE 网络的宏观架构。

为什么选 LTE 呢？首先，它是运营商在部署新网络时最可能考虑的。其次，可能也更重要的，就是 LTE 的特点之一就是架构简洁：组件更少、依赖更少，都能带来更好的性能。

7.4.1 无线接入网络（RAN）

RAN（Radio Access Network，无线接入网络）在任何运营商网络中都是第一重要的逻辑组件（图 7-10），它的主要任务是把请求转发到分配好的无线信道，从用户设备接收或者向用户设备发送数据。事实上，RAN 是一个由无线资源控制器（RRC，Radio Resource Controller）控制和管理的组件。在 LTE 中，每个无线基站（eNodeB）都安装有 RRC，负责维护 RRC 状态机并为小区内每个用户分配资源。

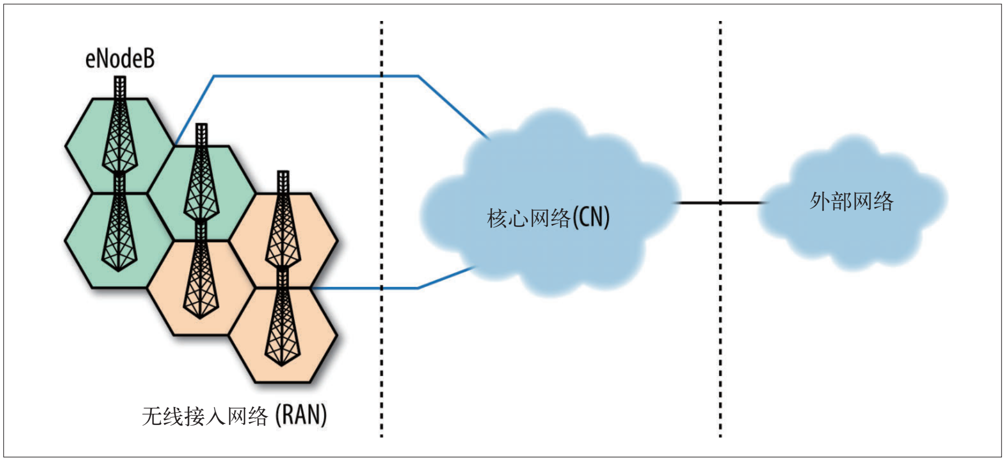


图 7-10：LTE 无线接入网络：跟踪小区及 eNodeB

只要邻近小区的信号更强，或者当前小区超载，用户就会被移交给邻近小区。说起来容易，事实上移交过程正是所有运营商网络中最复杂的环节。如果每个用户的位置固定，而且一个信号塔就可以覆盖，那么静态的路由拓扑就够了。可是，我们都知道，事实并非如此：用户是移动的，因此必然会在信号塔之间转移，而且转移期间不能影响语音和数据通信。不用说，这个问题可不容易解决。

如果用户的设备可以与任何无线信号塔关联，那我们怎么知道把接收到的数据包转发到哪里呢？当然，肯定没有魔法，答案很简单：无线接入网络必须与核心网络通信，以便随时知道每一位用户的行踪。更进一步，为了满足无干扰移交的需求，RAN 还必须能动态更新已有信道和路由，不打断用户发起的既有通信。

> 在 LTE 中，信号塔之间的移交可以在几百 ms 间完成，但也会导致物理层数据传输的短暂间断。但无论如何，用户对这个过程完全没有意识，对设备上正在运行的应用也没有影响。但在早期网络中，同样的过程需要花几秒钟时间。

可是，还不止如此。由于无线电移交频繁，特别是在高密度的市区和办公环境下，用户设备必须不断进行小区转移协商，即使设备处于空闲状态时也不例外，因此会消耗很多电能。事实上，这里又加入了一个中间层：一或多个无线信号塔构成了“跟踪区”，这是由运营商定义的一个逻辑上的信号塔群。

核心网络必须知道用户的位置，但更多时候，它只知道跟踪区，而不知道哪个信号塔正在为用户提供服务。稍后我们会提到，这对入站数据发送的延迟有重大影响。相应地，设备也因此得以在很低的耗电量下在同一跟踪区内转移：如果设备的 RRC

状态为空闲，设备或无线网络都不会发送通知，因而节约了电能。

7.4.2 核心网络

核心网络（图 7-11），也称为 EPC（Evolved Packet Core，演进分组核心网），在 LTE 中负责数据路由、账户和策略管理。简言之，它就是把无线网络和公共互联网连接到一起的部分。

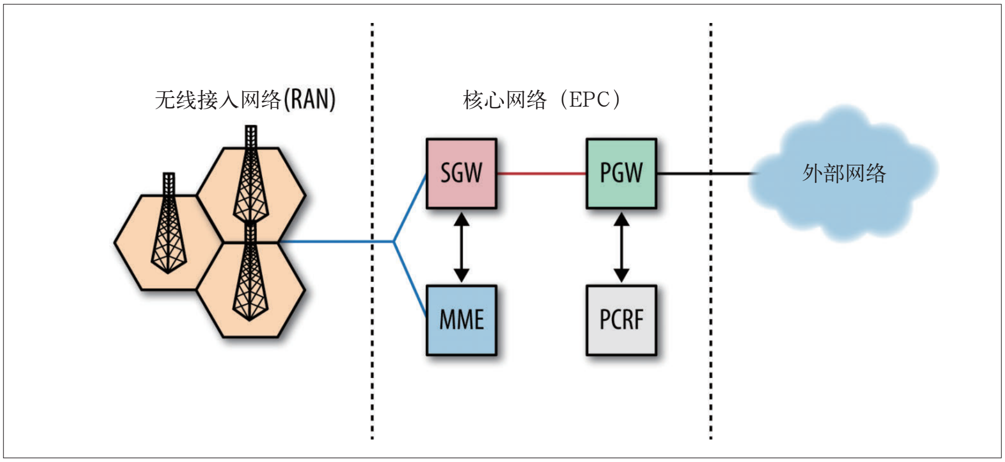


图 7-11：LTE 核心网络（EPC）：PGW、PCRF、SGW 和 MME

首先，就是分组网关（Packet Gateway）PGW，它负责连接移动运营商与公共互联网。PGW 是所有外部连接的终点，无论协议是什么。事实上，当移动设备连接到运营商网络时，正是 PGW 负责给它们分配和维护 IP 地址。

运营商网络中的每一部设备都有一个内部标识码，与被分配的 IP 地址无关。相应地，当 PGW 接收到一个分组，它会包装该分组并通过 EPC 发送到无线接入网络。LTE 对控制面流量使用 SCTP（Stream Control Transmission Protocol，流控制传输协议），对其他数据使用 GTP（GPRS Tunneling Protocol，GPRS 隧道协议）和 UDP。

物理层连接与应用层连接

由 PGW 分配和维护设备 IP 地址有一些重要的暗示。首先，这意味着无线设备很容易与多个 IP 地址关联。相反，如果 IP 地址非常珍贵，那么多个设备可以共享同一个 IP 地址，但分配不同的端口，以便对外收发数据。换句话说，PGW 在这里充当了 NAT 的角色！事实上，后一种情形相当常见。同一个运营商 IP 地址可以指定给网络内的几十甚至几百部设备。

结果，一部设备的流量可能源于多个公共运营商 IP 地址。一个客户端向多个 IP 地址请求资源没有什么值得大惊小怪的！到了 IPv6 时代，这种情况可能会发生变化，每部设备最终可能获得一个唯一的 IP 地址。话虽这么说，但支持 IPv6 的运营商还很少，IPv6 的推广和应用进程还很缓慢。

除了 IP 地址的分配之外，更重要的可能就是要理解：正因为终止外部连接的是 PGW，因此设备无线电模块的状态（空闲、活动、休眠），不会与任何外部连接的状态相关，无论是什么应用协议！

关闭无线网络中的无线信号，就可以断开设备与无线信号塔之间的物理链路。可是，TCP 和 UDP 等已经建立的高层连接仍然保持活动。在必须发送数据的时候，再重新建立物理连接，除了重新构建无线通信环境所需的 RRC 延迟，通信可以在没有任何副作用的情况下恢复。

PGW 还负责执行所有公共策略，比如分组过滤和检测、QoS 分配、DoS 保护等。PCRF（Policy and Charging Rules Function，策略和计费规则功能）组件负责维护和评估这些针对分组网关的规则。PCRF 是逻辑组件，即它可以内置于 PGW，也可以独立存在。

现在，我们假设 PGW 从公共互联网上接收到了一个分组，需要转发给其网络中的一部移动设备，那么它怎么路由该数据呢？ PGW 不知道用户的实际位置，也不知道无线接入网络中不同的跟踪区。下一步就该 SGW（Serving Gateway，服务网关）和 MME（Mobility Management Entity，移动管理实体）上场了。

PGW 把所有分组转发给 SGW，但糟糕的是，SGW 同样不知道用户的确切位置——这正是 MME 的一个核心任务。MME，即移动管理实体，实际上是一个用户数据库，管理着网络上所有用户的全部状态：他们在网络中的位置、账户类型、账单状态、启用的服务，以及所有用户元数据。只要用户在网络中的位置发生了变化，位置更新信息就会发送到 MME，当用户打开了自己的手机时，MME 也要负责确认身份。

实际上，当分组到达 SGW 时，SGW 就会向 MME 发送一个查询，查询用户的位置。MME 返回的用户位置信息包含跟踪区和为目标设备提供服务的特定信号塔的 ID，SGW 会利用该信息建立到信号塔的连接，然后将用户数据转发给无线接入网络。

简单来说，这些就是全部内容了。这个宏观的架构对几代移动数据网络而言都是相同的。其中逻辑组件的名字可能会不一样，但所有移动网络基本上都遵循下面这个工作流程。

- 数据到达外部的与互联网相连的分组网关。
- 为分组网络应用一套路由和分组策略。

- 将数据从公共网关路由到一或多个服务网关，这些服务网关是无线网络中设备的移动锚点。
- 通过用户数据库对网络中的每个用户进行身份验证、账单结算、服务提供及位置跟踪。
- 确定了无线网络中用户的位置后，用户数据就会从服务网关路由到相应的无线信号塔。
- 无线信号塔进行必要的资源分配并与目标设备协商，然后通过无线接口发送数据。

LTE 核心网络简化和统一的架构

LTE 的一个特点就是其新的 EPC 网络，该网络只基于 IP 在一个统一的网络中传输语音和数据。这种设计可以保证运营商的成本效率，同时也对网络的性能提出了更高的要求。语音必须低延迟，而 4G 速度的吞吐量也要更大。

EPC 怎么实现这些目标呢？当然，改进有很多，但无论如何，与前几代网络最主要的区别还是 LTE 核心网络的简化架构。特别是，为了实现更好的性能，删除了某些组件，另一些组件被合并为更少的逻辑组件，很多决策都被转移到了网络边缘。

比如，LTE 中的 RRC 由无线信号塔（eNodeB）维护，而在前几代网络中，RRC 都是由网络中的更高层（服务网关）来管理，这样也就在网络控制流量时引入了额外的延迟和性能问题。

7.4.3 回程容量与延迟

在任何运营商网络中，逻辑与物理组件之间的连接性及容量都是影响性能的重要因素。LTE 无线接口下的用户与信号塔之间可以达到 100 Mbit/s 的速度。但无线信号塔在接收到信号后，必须有足够的容量通过运营商网络将所有数据发送给实际的目标。此外，不要忘了一个信号塔应该能够同时服务于多个活动用户！

换句话说，真正的 4G 体验可不是部署一个新无线网络那么简单。核心网络也必须升级，EPC 和无线网络之间必须存在容量足够大的链路，而且 EPC 组件相较于前几代网络，必须能以更低的延迟处理更高的数据传输速率。

> 实践中，一个无线信号塔最多可以服务 3 个相邻小区，因此很容易就能累积到上百个活动用户。简单做一点数学计算，就能明白为了匹配 4G 无线网络的吞吐量，每个信号塔都必须有一条专用的光纤链路！

不用说，这些条件使得 4G 网络对运营商而言是一个昂贵的方案：所有无线基站都要接入光纤、高性能路由器，等等。实践中，经常能够看到网络整体性能受限于运营商网络的回程容量（backhaul capacity），而非无线接口。

这些性能瓶颈是我们作为移动应用开发人员无法控制的，但它们却再一次昭示了一个重要的事实：架构关系到端到端的性能。消除了第一跳（也就是无线接口）的瓶颈，也就是消除了网络中第二慢链路的瓶颈，无论是在运营商网络中，还是在通往目的地的其他路径上。

事实上，这也不是什么新鲜东西了，我们早在 1.4 节“延迟的最后一公里”中就提到过这个问题。连接到 4G 网络，并不意味着你能享受相应无线接口提供的最大吞吐量。相反，我们的应用必须适应运营商网络，乃至互联网上不断变化的网络条件。

7.5 移动网络中的分组流

在开发移动应用的过程中，人们抱怨最多的就是延迟时间捉摸不定。现在，既然我们已经了解了 RRC 和移动网络的宏观架构，接下来就可以把这些知识点联系起来，看一看端到端的分组数据流，从而揭示出延迟时间变化不定的原因。实际上，通过分析我们也会明白，其中很多可变性其实很大程度上还是可以预测的！

7.5.1 初始化请求

我们假设用户已经通过了 4G 网络的验证，移动设备处于空闲状态。现在，用户打开浏览器，输入 URL，点击“前往”。接下来会发生什么？

首先，由于手机处于空闲 RRC 状态，因此无线电模块必须与附近的信号塔同步，然后发送一个请求，以便建立无线通信环境（图 7-12，第①步），此次协商需要手机与信号塔之间的几次往返，可能需要花 100 ms。如果是前几代网络，那 RRC 由服务网关负责管理，这个协商过程要长得多，可达几秒钟。

建立了无线通信环境后，设备就会从信号塔得到相应资源，从而能够以特定的速度和功率传输数据（第②步）。用户数据从无线电模块传输到信号塔的时间称为用户面单向延迟，在 4G 网络中大约要花 5 ms。实际上，由于需要 RRC 切换，第一个分组花的时间比较多，但只要无线电模块保持高功率状态，后续分组的延迟时间都是恒定的，也就是第一跳的延迟时间。

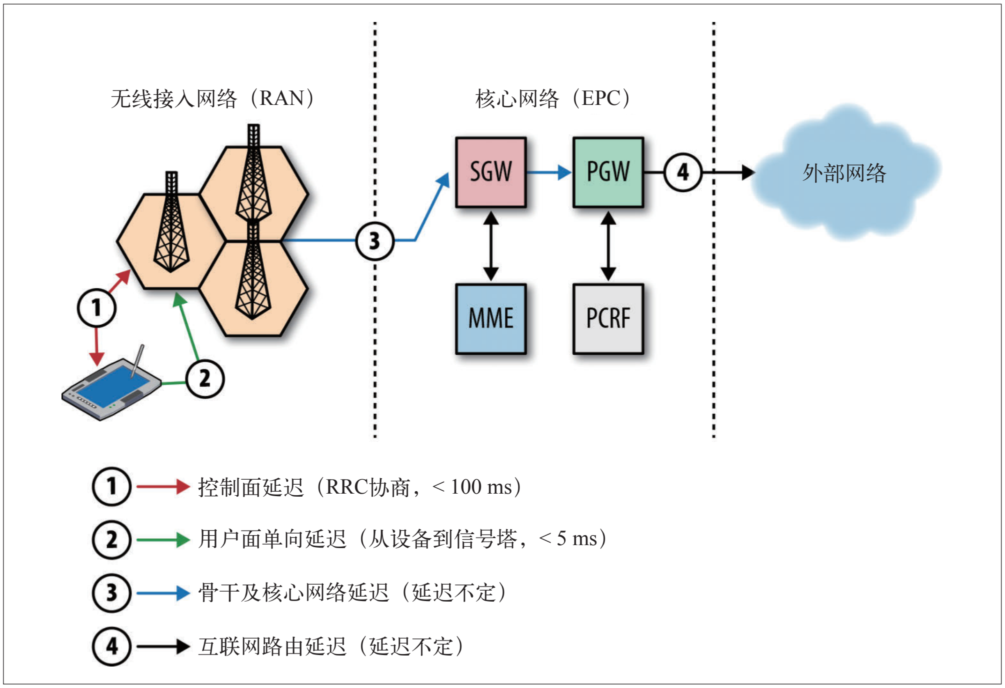


图 7-12：LTE 请求流的延迟时间

不过，还没有完，现在只是把分组从设备传输到了无线信号塔！然后，分组还要通过核心网络从 SGW 传输到 PGW（第③步），再向外传到公共互联网（第④步）。遗憾的是，这条路径上的延迟时间 4G 标准也无法保证，因运营商而异。

> 在很多已部署的 4G 网络中，只要设备处于连接状态，这种端到端的延迟一般为 30~100 ms。换句话说，不包含由最初的分组带来的控制面延迟。事实上，无线网络第一跳大约为 5 ms，其余时间（25~95 ms）就是在运营商核心网络中路由和传输的时间。

下面，再假设浏览器已经取得了请求的页面，用户正在浏览内容。无线电模块已经空闲了几十秒钟了，这意味着 RRC 很可能已经把用户切换到了 DRX 状态（见 7.3.2 节“LTE RRC 状态机”），从而节省电量，同时也释放网络资源供其他用户使用。此时，用户又想在浏览器中打开另一个页面，也就是触发了一个新请求。接下来又会发生什么呢？

大致的过程与前面所述是一样的，只不过因为设备处于休眠（DRX）状态，设备与无线信号塔间的协商速度要稍微快一些（图 7-12，第①步），即从休眠到连接不超过 50 ms（表 7-9）。

总之，用户发出的一次新请求总会导致一些不同的延迟。

- *控制面延迟*
  由 RRC 协商和状态切换导致的固定的、一次性的延迟时间，从空闲到活动少于 100 ms，从休眠到活动少于 50 ms。

- *用户面延迟*
  应用的每个数据分组从设备到无线电信号塔之间都要花的固定的时间，少于 5 ms。

- *核心网络延迟*
  分组从无线电信号塔传输到分组网关的时间，因运营商而不同，一般为 30~100 ms。

- *互联网路由延迟*
  从运营商分组网关到公共互联网上的目标地址所花的时间，可变。

前两个延迟时间都是 4G 标准规定的，核心网络延迟因运营商而异，最后一个延迟可以通过把服务器放到离用户较近的地方来缩短（参见 1.3 节“光速与传播延迟”）。

移动网络中的延迟与抖动

移动网络的主要问题其分组延迟的摇摆不定，或者叫抖动。没错，确实有一些相关组件会影响延迟。但是，知道了第一个分组触发 RRC 状态切换，从而导致的控制面延迟之后，你会发现性能比想象得要好预测得多。

在 LTE 中，控制面延迟最多 100 ms。而在 LTE Advanced 中，这个时间可以低至 50 ms。不过，在早期几代网络中，这个延迟可能达到几秒钟！

其次是核心网络延迟，这部分延迟在移动网络中经常占到分组总延迟的相当部分。具体的延迟时间因各代网络而不同，运营商的基础设施对核心网络延迟也会产生影响。虽然很少有运营商公开宣传自己的延迟时间（或许是因为没有什么值得炒作的），但你往往可以在“常见问题”中找到答案。

比如美国最大的移动运营商 AT&T，就为不同代网络的核心网络延迟给出了如下期望值，这些值在很大程度上代表了行业水平。

表7-10：AT&T部署的2~4G网络的延迟时间

	LTE	HSPA+	HSPA	EDGE	GPRS
延迟时间	40~50 ms	100~200 ms	150~400 ms	600~750 ms	600~750 ms

我们知道，地球赤道周长为 40 074 km，光绕地球一周需时 133.7 ms。也就是说，每一次移动请求平均至少要花光绕地球一周的时间！

7.5.2　入站数据流

下面再看一看相反的过程：用户设备处于空闲状态，但有一个数据分组要从 PGW 路由到用户（图 7-13）。没错，所有连接都终止于 PGW，因而设备才得以空闲（无线电模块关闭），但设备之前建立的连接（比如长 TCP 会话）可能在 PGW 中仍然是活动的。

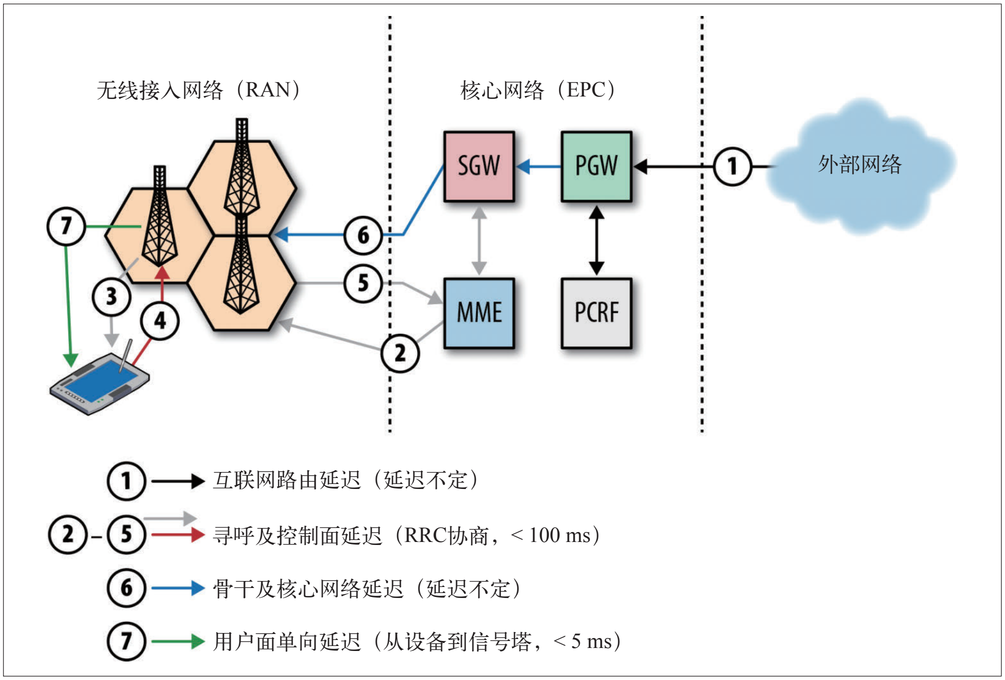


图 7-13：LTE 入站数据流延迟

如前所述，PGW 会把入站分组路由到 SGW（第①步），SGW 进一步查询 MME。然而，MME 可能不知道当前为用户服务的信号塔的确切位置（还记得吗，一堆信号塔构成一个“跟踪区”）。用户只要进入不同的跟踪区，其位置就会在 MME 中得到更新。但同一跟踪区内信号塔间的转移，不会触发 MME 的更新。

为此，如果设备处于空闲状态，MME 会向当前跟踪区内的所有信号塔发送一条寻呼消息（第②步），收到消息的信号塔接着通过共享的无线信道广播一条通知（第③步），告知设备应该重建无线通信环境，以便接收数据。设备周期性地唤醒以监听寻呼消息，如果在寻呼列表中发现了自己，它就会向无线电信号塔发送一条协商请求，请求重建无线通信环境。

无线通信环境重建之后，负责协商的信号塔向 MME 回发一条消息（第⑤步），表示它正在为用户服务。然后，MME 向服务网关返回一个应答，服务网关于是就会把数据路由到该信号塔（第⑥步），该信号塔再把数据转发给目标设备（第⑦步）。（还能咋样？）

设备一旦处于连接状态，无线信号塔与服务网关之间就会建立一条直连信道，从而让后续到达的数据能跳过第②～⑤步（不用再经过寻呼），直接被转发到信号塔。换句话说，仍然是第一个分组的延迟较长！必须记住。

> 对于 IP 及其上面的所有层，以及我们的应用而言，上述分组数据流是完全透明的。PGW、SGW 和 eNodeB 会在每一步负责缓冲分组，确保可以把它们送达设备。实践中，可以观察到分组到达时间中的延迟抖动，负责控制面协商的第一个分组导致的延迟最高。

7.6　异质网络（HetNet）

现有 4G 无线及调制技术已经接近无线信道的理论极限。事实上，下一个改进无线性能的研究方向不在无线接口，而在智能无线网络拓扑。具体来说，通过广泛部署多层异质网络（heterogeneous networks，HetNets），可以促进小区内协调、转移和干扰管理等多方面的改进。

HetNet 背后的核心思想非常简单：覆盖较大地理区域的无线网络容易导致用户竞争，因此不如用更小的小区来覆盖这些区域（图 7-14），实现每个小区的路径损失最小、传输功率最低，从而让所有用户享受到更高的性能。

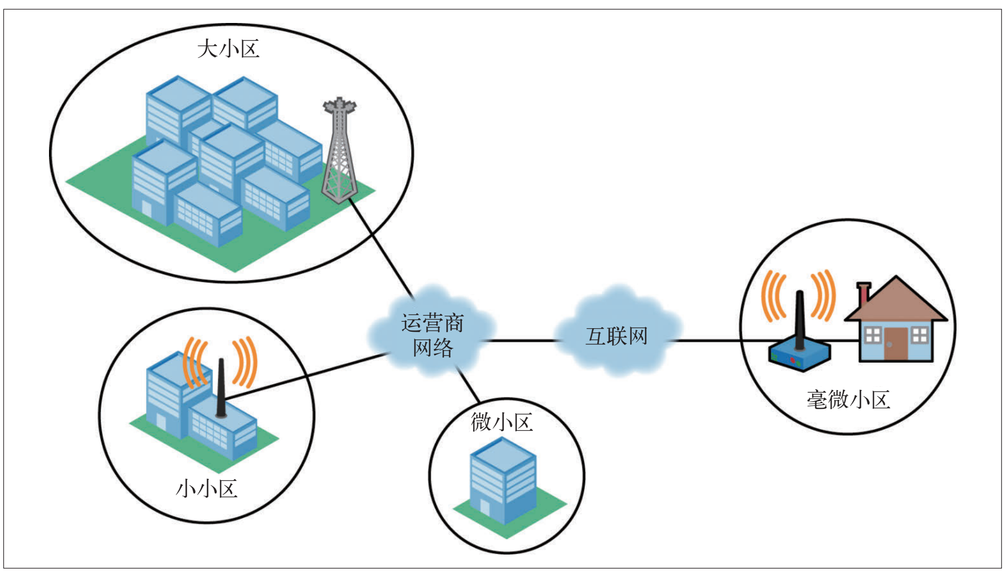


图 7-14：异质网络信息图（爱立信）

在低密度无线环境下，一个大小区可以覆盖几十公里，但在实践中，高密度的城市中心区和办公区，覆盖范围只能达到 50~300 m！换句话说，只能覆盖一个街区或

几栋楼。相对而言，小小区只覆盖某一栋楼，微小区则服务于几层楼，而毫微小区则覆盖一个小房间并利用现有频宽服务作为无线回程。

不过，HetNet 并不是简单地用大量小型小区替代大型小区，而是实现了小区的分层！通过部署层叠式无线网络，HetNet 能够提供更高的网络容量，覆盖更广泛的地理区域。HetNet 面临的挑战是如何降低干扰，提供足够的上行容量，以及制定和改进在不同网络层之间无缝迁移的协议。

这些对于我们开发移动应用的人来说意味着什么呢？意味着我们应该预见到不同小区之间的迁移次数会明显增多，因此要想办法适应。而且，由此带来的延迟及吞吐量性能变动也会非常明显。

建模和管理无线网络容量

移动运营商通常使用微小区把信号覆盖范围从室内扩展到信号质量较差的室外，或者在手机密集区（比如大型公共场所、会议厅、体育场、火车站，等等）用于扩充网络容量。有的微小区是永久部署的，而另一些可能只为特定场合部署——无线容量的规划与建模（图 7-15）既是艺术也是科学！

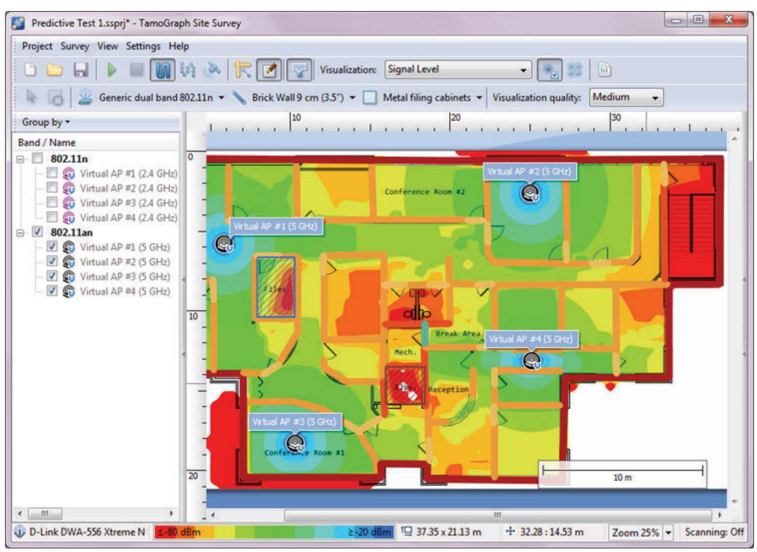


图 7-15：使用 TamoGraph 进行无线容量规划

图 7-15 中的 TamoGraph Site Survey 是专门用于建模的软件，通常用于对物理环境、活跃用户数量，以及可用的无线技术（图中的 Wi-Fi）建模，以辅助确定网络热点的必要数量、位置和配置。

7.7 真实的3G、4G和Wi-Fi性能

此时此刻，有读者可能禁不住想问：3G 或 4G 网络中的这些协议、网关和协商机制如此复杂真的有必要吗？你看 Wi-Fi，不是要简单得多吗，而且实践中似乎也用得很好啊！

这个问题可不好回答，正如我们前面介绍的，度量无线网络的性能离不开各种环境和技术性因素。更进一步，这个问题的答案还取决于选择什么样的评估标准：

- 电池性能重要，还是网络性能重要；
- 单用户性能，还是全网吞吐量性能；
- 延迟性能，还是分组抖动性能；
- 成本，还是部署的可行性；
- 满足政府及政策管制要求的能力；
- 其他很多标准。

不过，尽管干系方众多（用户、运营商、移动设备制造商，等等），而且各方都有自己的优先标准，最新 4G 网络的测试结果表明它仍然是非常有前景的。事实上，4G 在网络延迟、吞吐量和容量等这些关键指标上都胜过 Wi-Fi！

举一个具体的例子吧，美国密歇根大学和 AT&T 实验室的一个联合研究项目曾在美国境内做过测试，比较了 4G、3G 和 Wi-Fi（802.11g，2.4 GHz）的性能（图 7-16）。

- 通过 46 个独立的测量实验室节点测量性能，这些节点都是用于互联网测量的开放性平台，测量使用的是开源测量客户端 MobiPerf。
- 2011 年底，在两个月时间内，定期对 3300 名用户进行了测量。

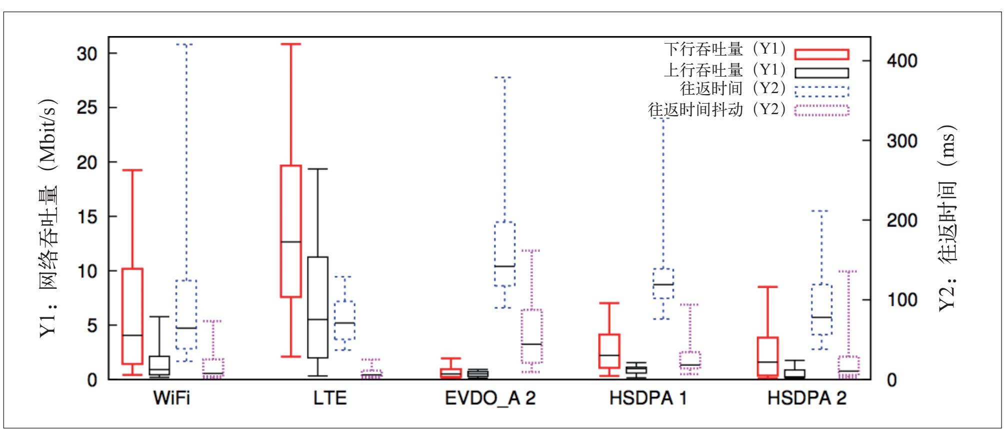


图 7-16：对 Wi-Fi、LTE 和 3G 性能的测量结果分析

每种连接类型的箱线图浓缩了很多有用的信息：线表示整体分布的范围，箱表示分布的下四分位和上四分位，箱中的黑色水平线是中位值。

当然，一次测试并不能找出普遍规律，特别是在关系到性能的情况下。但这个测试结果无论如何都很值得期待：早期的 LTE 网络展示出了巨大的吞吐量，而更让人兴奋的则是往返时间（RTT）和分组抖动时间相对于其他无线标准，明显更加稳定。

换句话说，这次测试至少表明，LTE 的性能好过 Wi-Fi，同时也表明性能提升是可能的，因此所有复杂性也就值得了！移动网络不一定慢。事实上，我们有理由相信，移动网络的速度还会更快。

要了解关于 4G 网络性能研究、分析和结论的更多信息，请参考在移动系统、应用及服务国际大会 MobiSys 2012 上发表的文章“A Close Examination of Performance and Power Characteristics of 4G LTE Networks”（关于 4G LTE 网络性能及功率特点的仔细研究）。

第 8 章
移动网络的优化建议

首先，通过持久连接、将服务器和数据部署到接近用户的地方、优化 TLS 部署，以及我们介绍的其他协议优化策略来降低延迟时间只会对移动应用更重要。当然，对移动应用而言，延迟时间和吞吐量都至关重要。类似地，所有 Web 性能最佳实践也同样适用，你现在就可以翻到第 10 章。

不过，移动网络也对我们的性能策略提出了新的、独特的要求。设计移动互联网应用，必须认真规划和考量设备的形态限制以展示内容，考虑无线接口的性能特性，以及电池使用时间。上述三个方面密不可分。

或许是因为表现层最容易控制，再加上响应式设计之类的概念，使得它抓足了人们的眼球。然而，大多数应用还是不尽人意，究其原因就是对网络性能的错误评估：应用的协议相同，但物理传输层的差别却有很多限制，如果对这些限制估计不足，就会导致响应速度慢、延迟时间摇摆不定，最终导致用户体验大打折扣。此外，非理性的联网操作也会对设备电池的使用时间造成极大的负面影响。

对于这三方面的限制，没有普适的解决方案。针对表现层、联网和电池使用时间，都有一些最佳实践，但这些做法经常相互冲突。最终，还是要靠你和应用在需求中取得平衡。但有一件事是肯定的：忽视其中任何一个限制都将对你不利。

我们不会详细讨论表现层，因为平台和应用类型对表现层的影响很大，而且讨论这方面的图书也已经有很多了。可是，无论制造商是谁，无论什么操作系统，移动网络的无线及电池限制都是普适的，因此这两方面将是本章讨论的重点。

> 本章，特别是接下来几页中，“移动应用”这个词将包含广泛的意义。我们所有关于移动网络性能的讨论，同样也适用于本机应用和浏览器应用，而且与平台及浏览器制造商无关。

8.1 节约用电

说到移动设备，节约用电是设备制造商、运营商、应用开发者，以及应用的用户都会关心的话题。在拿不准或不知道某个功能为什么存在或怎么实现时，不妨问自己一个简单的问题：这个功能对电量有什么影响，或者它怎么做到省电？事实上，这是一个对应用的所有功能都适用的问题。

移动网络的性能与电池使用时间天生联系在一起。而且，为了节约用电，无线接口的物理层还专门针对如下限制（或事实）做出了优化：

- 全功率打开无线电模块只消几小时就可耗尽电量；
- 对无线电功率的需求随着无线标准演进与代俱增；
- 无线电模块的耗电量仅次于设备的屏幕；
- 数据传输时无线电通信的耗电过程是非线性的。

知道了这些之后，在开发移动应用时，就该尽量少用无线电接口。当然，并不是说完全不能使用无线电模块，毕竟移动联网应用肯定还是要上网的！只是考虑到无线电通信直接关系到电池使用时间，我们应该尽最大可能在无线电开启时传输数据，而尽量把唤醒无线电以传输数据的次数减到最少。

> 虽然 Wi-Fi 也使用无线接口传输数据，但重要的是必须知道 Wi-Fi 的底层机制，及其延迟时间、吞吐量和耗电特点。这些与 2G、3G 和 4G 移动网络相比，都有根本不同（参见 7.3.1 节）。因此，应用联网时的行为自然也会因使用 Wi-Fi 或移动网络而有所不同。

使用 AT&T 应用资源优化器测量能源消耗

谁都知道应该节约使用电池，但当前大多数平台都没有给开发人员配备必需的工具，以测量和优化应用。可喜的是，我们还可以找到第三方工具，比如 AT&T 免费的 Application Resource Optimizer（ARO，应用资源优化器）工具包（图 8-1）。

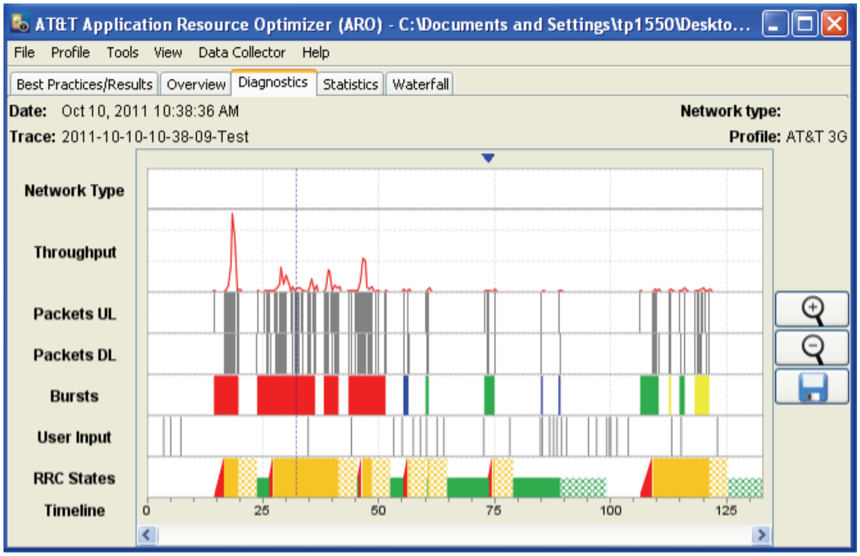


图 8-1：AT&T 应用资源优化器

ARO 包含两个组件：收集器和分析器。其中，收集器是一个后台 Android 应用（可以在手机或模拟器中运行），用于捕获传输的数据分组、无线模块活动信息及其他与手机的交互行为。要想记录用电情况，可以打开收集器，点击记录，使用应用，然后将记录结果复制到系统中。

得到记录结果后，可以通过分析器打开它，从而得知无线电状态、电量消耗、应用的通信模式等信息。另外，分析器有一个很不错的功能，即针对常见的性能陷阱提供建议，比如没有压缩、重复传输数据，等等。

有两点需要注意：电量消耗和无线电状态是通过设备及无线网络类型的特殊模型生成的。换句话说，生成的数值并不是设备在使用时的真实值，而是根据特定模型参数得到的估计值。从好的一面看，你可以利用它导入不同的设备和网络模型，比较不同模型（比如 3G、4G）下的电量消耗。

第二点，收集器只能在 Android 平台运行，而 ARO 分析器则可以接收任何常规的分组跟踪记录（pcap）文件，可以使用 `tcpdump` 或其他兼容工具生成；iOS 用户可以使用 `tcpdump`。

要了解 ARO 工具包，可以访问 developer.att.com/ARO/OreillyHPBN。

8.2 消除周期性及无效的数据传输

我们知道，无论传输多少数据，移动无线通信总会消耗恒定电量，从而导致无线模块进入高功率状态。因此，对于耗电而言，根本不存在什么“耗电少的请求”（参见 7.3.5 节“低效率的周期性传输”）。那么，进一步归纳就可以得到如下规则：

- 轮询在移动网络中代价极高，少用；
- 尽可能使用推送和通知；
- 出站和入站请求应该合并和汇总；
- 非关键性请求应该推迟到无线模块活动时进行。

一般来说，推送比轮询效果更好。但频率过高的推送与轮询也不相上下。如果碰到实时更新的需求，应该考虑下列问题。

- 最佳更新间隔多长，是否符合用户预期？
- 除了固定的更新间隔，能否因地因时制宜？
- 入站或出站请求能否集合为更少的网络调用？
- 入站或出站请求能否推迟到以后发送？

> 对推送而言，原生应用可以访问平台专有的推送服务，因此应该尽可能使用。对 Web 应用来说，可以使用 SSE（Server Sent Events，服务器发送事件）和 WebSocket 以降低延迟时间和协议消耗，尽可能不使用轮询和更耗资源的 XHR 技术。

基于时间间隔、环境、用户偏好，甚至电池电量，简单地把多个通知集合到一个推送事件中，可以显著提升任何应用的电池效率。后台应用尤其适合这一策略，因为它们经常要以这种方式访问网络。

内格尔及有效的服务器推送

因为有内格尔（Nagle）算法，TCP 的拥趸一定会认可请求聚合和绑定的建议，当然不是在应用层！内格尔算法尝试将多个 TCP 消息组合为一个分组，以期减少协议消耗和需要传输的分组数量。不用说，移动应用同样是这一技术的用武之地。

简单的策略是，可以在服务器上根据时间、数量或大小来聚合消息，而不是每个消息都单独推送一次。但更深入也更有效的做法，则是只在客户端的无线模块活动期间推送更新。比如，把消息推迟到客户端发送请求之后再发送，可以利用那些能够监控客户端无线状态的服务。

举个例子，GCM（Google Cloud Messaging，谷歌云消息）就是一个可以在 Android 和 Chrome 平台中使用的消息发送 API，它能聚合消息，并只在设备活动时发送更新。只要把消息发送给 GCM，然后 GCM 就能作出最优的推送时间安排。

遗憾的是，目前还没有功能类似 GCM 的跨浏览器 API，因此不能统一为所有客户端提供服务。不过，W3C Push API（参见 www.w3.org/TR/push-api/）将来有可能解决这个问题。

间歇的信标（beacon）请求（比如周期性的听众评测和实时分析请求）很容易破坏电池电量优化策略。这些周期性的请求对于有线甚至 Wi-Fi 网络没有什么负面影响，但对无线网络影响巨大。这些信标请求有必要实时发送吗？恐怕借助日志将请求推迟到无线模块下一次活动时发送才是明智的。总之，需要认真对待后台的这些周期性操作，同时还要密切关注第三方库和代码片段访问网络的模式。

最后一点，虽然我们一直在说电池，但也不要忘了渐进增强和增量加载等技术依赖的间歇性网络访问，会带来较长的延迟，因为 RRC 状态需要切换！还记得每次状态切换都会导致移动网络控制面的较大延迟吧，每次切换都可能增加几百甚至几千 ms 的延迟时间。对于用户发起或交互性的流量，这方面的代价非常之大。

计算后台更新的电量消耗

为说明周期性轮询对电池使用时间的影响，我们可以做一道简单的数学题。数值虽然不一定准确，但也不会跑出 3G/4G 移动设备的范围之外：

- 5 Wh 或 18000 J 的电池容量（5 Wh × 3600 J /Wh）；
- 无线电模块从空闲状态切换到连接状态需要 10 J；
- 1 分钟一次的轮询，1 小时耗能 600 J（60 × 10 J）；
- 600 J 相当于电池容量的大约 3%（600 J / 18000 J）。

一个应用每小时就要消耗 3% 的电量！那么两个应用不重合的轮询，要耗尽电池，只需半天时间。当然，如果应用频繁使用不缓冲的推送（比如，每两秒就推送一次），那么其耗电量还会更高！

电池使用时间优化和更新频率天生是一对矛盾。实践中，要根据自己应用的特定需求来确定优化策略：更新打包、适应性的更新间隔、拉与推结合，等等。当然，还要使用 ARO 或类似的工具测量策略效果，适时调整。

消除不必要的长连接

TCP 或 UDP 连接的连接状态及生命期与设备的无线状态是相互独立的。换句话说，

即便与运营商网络仍维持着（两端间）连接不中断，无线模块也可以处于低耗电状态。外部网络的分组到来时，运营商无线网络会通知设备，使其无线模块切换到连接状态，从而恢复数据传输。

明白了吗，应用不必让无线模块“活动”也可以保持连接不被断开。但不必要的长连接也有可能极大地消耗电量，而且由于人们对移动网络无线通信的误解，这种情况经常发生。这里可以参考 7.4.2 节中的“物理层连接与应用层连接”和 7.5 节“移动网络中的分组流”。

> 大多数移动运营商的 NAT 连接超时时间为 5~30 分钟。因此，为了在空闲状态下保持连接，可能需要周期性（每 5 分钟）发送一次 Keep-Alive。如果你觉得自己的 Keep-Alive 发送太过频繁，需要先检查自己的服务器、代理和负载均衡器配置！

8.3 预测网络延迟上限

在移动网络中，一个 HTTP 请求很可能会导致一连串长达几百甚至上几千 ms 的网络延迟。这一方面是因为有往返延迟，另一方面也不能忘记 DNS、TCP、TLS 及控制面的延迟（图 8-2）。



图 8-2：一个“简单的”HTTP 请求的构成

最好的情况，就是无线模块处于最高功率状态、DNS 已经提前解析完成，而且 TCP 连接是现成的（客户端可以重用已有的连接，不用再花时间去建立新连接）。可是，如果连接繁忙，或不存在，那就必须在发送数据前经历额外的往返。

为说明额外的网络往返有多大影响，我们假设理想情况下 4G 网络的往返时间为 100 ms，3.5G+ 网络的往返时间为 200 ms。

在 3G 网络中，仅 RRC 控制面延迟一项就可以给重建无线通信环境增加几百到几千 ms 的延迟！无线模块活动后，可能还需要解析主机名和 IP 地址，然后进行 TCP 握手，这又是两次往返。然后，如果需要安全信道，可能还需要额外两次网络往返（参见 4.3 节）。最后，才能发送 HTTP 请求，这最少又是一次往返（表 8-1）。

表8-1：一个HTTP请求的延迟时间

	3G	4G
控制面	200~2500 ms	50~100 ms
DNS 查询	200 ms	100 ms
TCP 握手	200 ms	100 ms
TLS 握手	200~400 ms	100~200 ms
HTTP 请求	200 ms	100 ms
总延迟	**200~3500 ms**	**100~600 ms**

就这还没有考虑服务器响应时间、响应大小呢，这些也需要几次往返，最多可能达到 6 次。再乘以往返时间，那么 3G 网络的延迟可能长达数秒，4G 网络大约也要半秒！

8.3.1 考虑RRC状态切换

如果移动设备已经空闲了几秒钟，那应该假设并预想第一个分组将会导致几百甚至几千 ms 的额外 RRC 延迟。经验表明，4G 网络会增加 100 ms，3.5G+ 网络会增加 150~500 ms，而 3G 网络会增加 500~2500 ms 一次性的控制面的延迟。

7.3 节介绍的无线电资源控制器（RRC）是专门设计用来解决高耗电问题的。但在保证电池使用时间的同时，各种计时器、计数器的存在，以及不同无线状态切换所需的网络协商，也带来了额外的延迟和较低的吞吐量。然而，RRC 在移动网络中是事实存在的，无法回避的。如果你想针对移动网络优化自己的应用，必须在设计时就考虑到 RRC。

关于 RRC，以下是我们已经了解的一些常识：

- RRC 状态机因无线标准不同而不同；
- RRC 状态机由无线网络为每部设备分别管理；
- RRC 在必须传输数据时会切换到高功率状态；
- RRC 在网络配置的时间超出后会切换到低功率状态；
- （4G）LTE 状态切换可能花 10~100 ms；
- （4G）HSPA+ 状态切换与 LTE 相差无几；
- （3G）HSPA 和 CDMA 状态切换可能花几秒钟；
- 每次网络传输，无论数据多少，都会导致能量尾。

我们已经讨论了为什么对移动应用而言，节约用电是一个非常重要的目标。而且，我们也重点介绍了间隙传输的低效率，它又是因 RRC 状态切换而超时直接导致的结果。不过，还有一点需要注意：如果设备的无线模块已经空闲，那么在移动网络

上再次传输数据将导致额外的延迟，这个延迟在最新的网络上可能有几百 ms，而在 3G 或 2G 网络上最多则可能达到几秒。

换句话说，尽管网络会给人一种我们的应用永远在线的错觉，但由 RRC 控制的物理层（也就是无线模块）会不断连接和断开。表面上看这不是什么问题，但这种由 RRC 交互导致的延迟，如果不加重视的话，很可能会打断用户体验。

8.3.2　解耦用户交互与网络通信

设计得好的应用，即便底层连接慢或者请求时间长，通过在 UI 中提供即时反馈也能让人觉得速度快。不要把用户交互与网络通信联系得太过紧密。为给用户最佳体验，应用必须在几百 ms 内响应输入，具体参见 10.2.1 节“速度、性能与用户预期”。

如果必须发送网络请求，那么在后台发，但要对用户输入立即给出反馈。单单控制面延迟一项，经常就可以让你提供实时反馈的计划超出预算。因此要针对高延迟做足准备，虽然不能“解决”核心网络和 RRC 带来的延迟，但你可以与设计团队合作，确保他们在设计应用时了解这些限制。

8.4　面对多网络接口并存的现实

用户不喜欢速度慢的应用，但由于短暂网络错误导致的应用崩溃才是体验最差的。我们的移动应用必须足以应对各种常见的网络错误：无法访问的主机、吞吐量突然下降或延迟突然上升，甚至连接彻底断开。与有线网络不同，你不能假定一次连接成功就能持续保持连接状态。用户可能正在移动，可能进入了高冲突、用户多，或者信号差的区域。

另外，就像设计界面时不能只考虑最新的浏览器一样，设计应用时也不能只考虑最新一代移动网络。如前所述（7.1.7 节），即便用户手里拿着最新的手机，也需要不断在 4G、3G，甚至 2G 网络之间切换。我们的应用必须接受这些接口变化，作出相应调整。

> 在浏览器中，可以使用 `navigator.onLine` 接收连接状态通知，也可以利用 NetInfo（Network Information API，网络信息 API）查询和监听连接属性的变化。要了解更多信息，请参考 Paul Kinlan 在 HTML5 Rocks 上的文章：Working Off the Grid with HTML5 Offline（http://hpbn.co/offline）。

移动网络中不变的只有变化。距离信号塔的远近、活跃用户的多少、环境冲突、一天中的时段，以及其他很多不可预知的因素，都会导致移动信道质量的差异。明白

了这些，你就会知道为优化移动应用而对各种带宽和延迟所作的估算结果，最多就是一种瞬态数据点。

> iPhone 4 的“天线门”是无法准确预测无线性能的一个典型事例：信号质量会受到你握持手机的手与天线距离的影响。“你拿手机的姿势不对”这句流行语正是由此而来。

移动网络的延迟和带宽估计值基本上就是几十或几百 ms，最多 1 s。实际上，虽然 6.4.2 节中“自适应比特流”那一部分介绍的优化策略，即将数据分成多个几秒钟的块，对持久数据流（比如视频）还是有用的，但对带宽的估计值绝对不该缓存，或者用于决定将来的吞吐量，就算是对 4G 网络也不行！可能你刚刚测量的速度是几百 Kbit/s，而换了只手速度就到了 Mbit/s 以上。

移动网络中的流媒体应用

流媒体应用在移动网络中是一个难题。如果你需要一次长时间的下载，而且知道要用到整个文件，就应该一次性下载该文件，然后让无线模块尽量空闲更长时间。前面介绍的 Pandora 应用中下载音乐文件的例子就是一个典型。

可是，如果由于文件太大，或用户行为受限，不能一次性下载完整的文件（比如高清视频），那就应该利用 6.4.2 节中“自适应比特流”那一部分介绍的优化策略，适应网络吞吐量的不断变化。此时，电池耗电可能更快，但至少你能保证最佳用户体验！当然，也可以建议用户切换到 Wi-Fi 网络。

在任何网络中，要估计端到端的带宽和延迟都很难，移动网络尤甚。可以说，你的估计十有八九都不靠谱。相反，应该基于相关网络属于哪一代的粗粒度信息，相应地调整代码。记住，知道移动网络属于哪一代或者是什么类型，不能保证任何端到端的性能，但你却能知道无线网络第一跳的延迟数据，以及运营商网络中端到端的数据。这一块的详细信息，请参考 7.5.1 节中的“移动网络中的延迟与抖动”，以及表 7-6。

最后，除了吞吐量和延迟之外，还应该考虑连接中断。要把连接中断作为常态，而不是例外。不管网络是不是可用，你的应用都应该尽量保持运行，而且应该根据请求类型和特定的问题作出反应：

- 不要缓存或试图猜测网络状态；
- 调度请求、监听并诊断错误；
- 瞬态错误总会发生，不可忽视，可以采取重试策略；

- 监听连接状态，以便采用最佳请求方式；
- 对重试请求采用补偿算法，不要永远循环；
- 离线时，尽可能记录并在将来发送请求；
- 利用 HTML5 的 AppCache 和 localStorage 实现离线应用。

随着 HetNet 越来越多地得到部署，转换小区的频率将大幅上升，监控连接状态和类型只会变得更加重要。不过也有一个好消息，即较小的小区也应该提供较理想的总体吞吐量和延迟时间。

8.5 爆发传输数据并转为空闲

移动无线接口专门为爆发性传输做过优化，这一点应该尽可能利用。比如，把请求分组，尽可能多和快地下载数据，然后让无线模块转为空闲。这样，既可以获得最大的网络吞吐量，也能节约电量。

估算网络速度唯一正确的方式，就是使用它！ LTE 和 HSPA+ 等最新一代网络，每隔 1 ms 就会动态分配一次资源，而且更适合爆发性的数据传输。换句话说，要想快，就要简单：批量请求，预先下载尽可能多的数据，然后让网络空闲。

基于此，一个重要的结论就是：渐进加载资源在移动网络中弊大于利。每次只下载一点数据会导致应用的吞吐量和延迟都摇摆不定，同时消耗的电量可能也会更多。因此，要尽可能预先下载数据，预测用户接下来可能需要看什么，提前下载，尽量让无线模块空闲：

- 如果需要大型音频或视频文件，考虑提前下载整个文件，而不要以比特为单位地流式下载；
- 预先取得应用内容，通过测量和统计工具来辨别什么内容适合提前下载；
- 预先取得第三方内容，比如广告，通过应用逻辑提前显示并更新它们的状态；
- 允许设备关闭无线模块，保持其空闲，不要忘了优化和消除间歇性传输（参见 7.3.5 节中的“46% 的电量消耗仅传输 0.2% 的数据”）。

构建和评估预取模型

预取内容总会存在矛盾：一方面，你希望尽可能下载更少的数据，另一方面，你又想减少延迟和吞吐量的变动，同时降低对电池的影响。哪一方面更重要呢？这样问本身就是错误的。答案永远与应用的具体使用情境，以及你选择用来评估预取策略有效性的指标相关。

重点在哪里？最低限度，要做到三个变量的平衡：传输的字节数、对电池的影响，还有网络吞吐量及延迟的变动。而且，如前所述，这三个变量并不相互排斥。一次下载较多字节可能会带来更大的吞吐量！

对于能准确预测使用模式的应用，可以采取激进的预取策略，将电量消耗降到最低，提升用户体验，同时避免大下载量的开销。相反，不够好的预取策略可能会下载大量不必要的数据，降低整体用户体验。

要确定应用的具体行为，首先要确定你的主要目标，以及应用的主要使用模式。然后，根据这些因素决定预取策略，并收集数据以验证你的预测，如此反复。

8.6 把负载转移到Wi-Fi网络

目前的行业预测显示，世界范围内几乎 90% 的无线流量都源自室内，而且经常是在有 Wi-Fi 连接的区域内。虽然最新 4G 网络的峰值吞吐量和延迟时间都与 Wi-Fi 不相上下，但每月的数据流量往往都有上限，毕竟移动上网是按量计费的，价格并不便宜。另外，Wi-Fi 连接下的大数据量传输更省电（参见 7.3.1 节“3G、4G 和 Wi-Fi 对电源的要求”），也不需要 RRC。

可能的情况下，特别是对需要处理较大数据量的应用，都应该利用 Wi-Fi 连接。如果检测不到 Wi-Fi 连接，可以建议用户打开 Wi-Fi 连接，以提升体验和节省电量。

8.7 遵从协议和应用最佳实践

网络基础设施的分层架构有一个最大的优点，那就是把物理交付接口从传输层中抽象了出来，而传输层又把路由和数据交付从应用协议中抽象了出来。这种分离的结果就是 API 具有独立性，但为了取得端到端的最佳性能，我们仍然要考虑整个架构。

本章主要讨论了移动网络物理层的独立性能，比如 RRC 的存在、电池使用时间的问题，以及移动网络中独有的路由延迟。在物理层之上，是我们前几章已经介绍过的传输和会话协议，针对它们的优化同样甚至更加重要：

- 2.5 节“针对 TCP 的优化建议”；
- 3.3 节“针对 UDP 的优化建议”；
- 4.7 节“针对 TLS 的优化建议”。

通过重用持久连接、将服务器和数据部署到离客户端更近的地方、优化 TLS 部署，以及其他所有优化措施，对移动网络而言只会更加重要，因为移动网络的往返时间

长，而带宽永远都很昂贵。

当然，我们的优化策略并不会止步于传输和会话协议，这两方面只是基础。从现在开始，我们还必须考虑不同应用协议（HTTP 1.0、1.1 和 2.0），以及通用 Web 应用开发最佳实践对性能的影响。继续阅读吧，还没有结束呢！

第三部分

HTTP

第 9 章

HTTP简史

HTTP（HyperText Transfer Protocol，超文本传输协议）是互联网上最普遍采用的一种应用协议，也是客户端与服务器之间的共用语言，是现代 Web 的基础。从最初的一个关键字和文档路径开始，HTTP 最终不仅成为了浏览器的协议，而且也几乎成为了所有互联网软件和硬件应用的协议。

本章将简略回顾一下 HTTP 协议的发展史。全面探讨 HTTP 的各种语义不是本书的意图，但理解 HTTP 在设计上的关键转变，以及每次转变背后的动机——特别是 HTTP 2.0 将带来的很多改进，对我们讨论 HTTP 性能则至关重要。

9.1 HTTP 0.9：只有一行的协议

Tim Berners-Lee 最初的 HTTP 建议是以简洁为出发点设计的，目的是推动他的另一个刚刚萌芽的思想——万维网的应用。事实证明，这个策略非常有效。这个经验也非常值得有抱负的协议设计者汲取。

1991 年，Tim Berners-Lee 概述了这个新协议的动机，并罗列了几条宏观的设计目标：支持文件传输、能够请求对超文本文档的索引搜索、格式化协商机制，以及能够把客户端引导至不同的服务器。为了实际验证这个理论，他构建了一个简单的原型，实现了建议的部分功能：

- 客户端请求是一个 ASCII 字符串；
- 客户端请求由一个回车符（CRLF）结尾；

- 服务器响应是一个 ASCII 字符流；
- 服务器响应的是一种超文本标记语言（HTML）；
- 连接在文档传输完毕后断开。

然而，即便这样说也比实际情况复杂。以上规则定义了一个极其简单、可以通过 Telnet 验证的协议，某些 Web 浏览器直到今天仍然支持：

```
$> telnet google.com 80

Connected to 74.125.xxx.xxx

GET /about/

(超文本响应)
(连接关闭)
```

请求只有一行，包括 GET 方法和要请求的文档的路径。响应是一个超文本文档，没有首部，也没有其他元数据，只有 HTML。这实在是简单得不能再简单了！鉴于以上交互行为只实现了部分预期目标，因此相应的协议也被非官方地称为 HTTP 0.9。此后发生的事，大家都知道了。

以 1991 年这个低调开端为起点，HTTP 在随后几年中展现了自己的生命力，得到了迅速发展。下面我们简单总结一下 HTTP 0.9 的功能：

- 客户端 / 服务器、请求 / 响应协议；
- ASCII 协议，运行于 TCP/IP 链接之上；
- 设计用来传输超文本文档（HTML）；
- 服务器与客户端之间的连接在每次请求之后都会关闭。

Apache 和 Nginx 等流行的 Web 服务器至今仍然支持 HTTP 0.9，部分原因是支持它不费什么事儿！如果你感到好奇，可以打开 Telnet 终端，通过 HTTP 0.9 访问 google.com 或其他你熟悉的网站，观察一下这个早期协议的行为和局限性。

9.2 HTTP 1.0：迅速发展及参考性RFC

1991 年到 1995 年，HTML 规范和一种新型的名叫“Web 浏览器”的软件都获得了快速发展。与此同时，面向消费者的公共互联网基础设施，也日渐兴起并迅速发展起来。

完美风暴：1990 年代初期互联网的繁荣

在 Tim Berner-Lee 最初提出的浏览器原型基础上，NCSA（National Center of Supercomputer Applications，美国国家超级计算机应用中心）决定实现自己的版本。于是，世界上第一款流行的浏览器 NSCA Mosaic 诞生了。1994 年 10 月，NCSA 浏览器开发团队中的一员 Marc Andreessen，与 Jim Clark 合伙创建了 Mosaic Communications 公司。这家公司后来改名为 Netscape，并于 1994 年 12 月开始销售 Netscape Navigator 1.0。此时此刻，万维网已经在学术研究领域之外迈出很大步伐。

事实上，主导成立万维网联盟（W3C）的 CERN（Conseil Européen pour la Recherche Nucléaire，欧洲核子研究委员会）也在 1994 年组织召开了第一届万维网大会。而 IETF（Internet Engineering Task Force，互联网工程任务组）也成立了 HTTP 工作组（HTTP-WG），致力于改进 HTTP 协议。从那时起，这两个组织就对万维网的发展开始发挥作用，而且今后还将继续担当这一角色。

同样在 1994~1995 年，CompuServe、AOL 和 Prodigy 也开始提供拨号上网服务。于是，学术界和产业界合作演绎了一场“完美风暴”。就在这一波互联网浪潮的基础上，Netscape 的 IPO 于 1995 年 8 月 9 日获得了历史性成功。互联网火了，每个人都想从中分一杯羹。

随着人们对新兴 Web 的需求越来越多，以及它们在公共 Web 上的应用迅速爆发，HTTP 0.9 的很多根本性不足便暴露出来。人们需要一种协议，它不仅能访问超文本文档，还能提供有关请求和响应的各种元数据，而且要支持内容协商，等等。相应地，新兴的 Web 开发者社区为满足这些需求，推出了大量实验性的 HTTP 服务器和客户端实现，基本上遵循实现、部署、推广采用的流程。

就在这些实验性开发的基础上，出现了一套最佳实践和共用模式。于是，1996 年，HTTP 工作组发布了 RFC 1945，解释说明了当时很多 HTTP 1.0 实现的“公共用法”。不过，据我们所知，这个 RFC 只是参考性的。HTTP 1.0 并不是一个正式的规范或互联网标准！

不管怎样，HTTP 1.0 的请求对我们而言应该是非常熟悉的：

```
$> telnet website.org 80

Connected to xxx.xxx.xxx.xxx

GET /rfc/rfc1945.txt HTTP/1.0 ❶
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Accept: */*

HTTP/1.0 200 OK ❷
```

```
Content-Type: text/plain
Content-Length: 137582
Expires: Thu, 01 Dec 1997 16:00:00 GMT
Last-Modified: Wed, 1 May 1996 12:45:26 GMT
Server: Apache 0.84

(纯文本响应)
(连接关闭)
```

❶ 请求行中包含 HTTP 版本号，随后是请求首部
❷ 响应状态，后跟响应首部

上面列出的交换信息并未展示出 HTTP 1.0 的所有功能，但却能说明该协议的关键变化：

- 请求可以由于多行首部字段构成；
- 响应对象前面添加了一个响应状态行；
- 响应对象也有自己的由换行符分隔的首部字段；
- 响应对象不局限于超文本；
- 服务器与客户端之间的连接在每次请求之后都会关闭。

请求和响应首部都使用 ASCII 编码，但响应对象本身可以是任何类型：HTML 文件、纯文本文件、图片，或其他内容类型。事实上，HTTP 中的“HTT”（Hypertext Transfer，超文本传输）在协议出现后不久就已经用词不当了。在实践中，HTTP 迅速发展为超媒体传输协议，但最初的名字则沿用至今。

除了媒体类型协商，RFC 还解释了很多已经被实现的其他功能：内容编码、字符集支持、多部分类型、认证、缓存、代理行为、日期格式，等等。

> 今天，几乎所有 Web 服务器都支持，而且以后还会继续支持 HTTP 1.0。除此之外，剩下的你都知道了。但 HTTP 1.0 对每个请求都打开一个新 TCP 连接严重影响性能，这一点可以参考 2.1 节“三次握手”和 2.2.2 节“慢启动”。

9.3 HTTP 1.1：互联网标准

将 HTTP 转为 IETF 正式互联网标准的工作，与通过 RFC 1945 说明解释 HTTP 1.0 是并行展开的，从 1995 年到 1999 年，大致经历了 4 年时间。事实上，就在 HTTP 1.0 发布大约 6 个月之后，也就是 1997 年 1 月，定义正式 HTTP 1.1 标准的 RFC 2068 也发布了。又过了两年半，即到了 1999 年 6 月 RFC 2616 发布，又在标准中集合了很多改进和更新。

HTTP 1.1 标准厘清了之前版本中很多有歧义的地方，而且还加入了很多重要的性能优化：持久连接、分块编码传输、字节范围请求、增强的缓存机制、传输编码及请求管道。

有了这些新功能，我们就可以像任何当前的 HTTP 浏览器和客户端一样检视 HTTP 1.1 会话：

```
$> telnet website.org 80

Connected to xxx.xxx.xxx.xxx

GET /index.html HTTP/1.1 ❶
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ❷
Server: nginx/1.0.11
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Via: HTTP/1.1 GWA
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache
Transfer-Encoding: chunked

100 ❸
<!doctype html>
(snip)

100
(snip)

0 ❹

GET /favicon.ico HTTP/1.1 ❺
Host: www.website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: */*
Referer: http://website.org/
Connection: close ❻
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ❼
Server: nginx/1.0.11
```

```
Content-Type: image/x-icon
Content-Length: 3638
Connection: close
Last-Modified: Thu, 19 Jul 2012 17:51:44 GMT
Cache-Control: max-age=315360000
Accept-Ranges: bytes
Via: HTTP/1.1 GWA
Date: Sat, 21 Jul 2012 21:35:22 GMT
Expires: Thu, 31 Dec 2037 23:55:55 GMT
Etag: W/PSA-GAu26oXbDi

(图标数据)
(关闭连接)
```

❶ 请求 HTML 文件，及其编码、字符集和元数据

❷ 对原始 HTML 请求的分块响应

❸ 以 ASCII 十六进制数字表示的分块数据的字节数（256 字节）

❹ 分块数据流响应结束

❺ 在同一个 TCP 连接上请求图标文件

❻ 通知服务器不再使用连接了

❼ 图标响应，随后关闭连接

啊，这一次可复杂多了。首先，最明显的差别是这里发送了两次对象请求，一次请求 HTML 页面，一次请求图片，这两次请求都是通过一个连接完成的。这个连接是持久的，因而可以重用 TCP 连接对同一主机发送多次请求，从而实现更快的用户体验，参见 2.5 节“针对 TCP 的优化建议”。

为终止持久连接，客户端的第二次请求通过 Connection 首部，向服务器明确发送了关闭令牌。类似地，服务器也可以在响应完成后，通知客户端自己想要关闭当前 TCP 连接。从技术角度讲，不发送这个令牌，任何一端也可以终止 TCP 连接。但为确保更好地重用连接，客户端和服务器都应该尽可能提供这个信息。

> HTTP 1.1 改变了 HTTP 协议的语义，默认使用持久连接。换句话说，除非明确告知（通过 Connection: close 首部），否则服务器默认会保持连接打开。
>
> 不过，这个功能也反向移植到了 HTTP 1.0，可以通过 Connection: Keep-Alive 首部来启用。实际上，如果你使用的是 HTTP 1.1，从技术上说不需要 Connection: Keep-Alive 首部，但很多客户端还是选择加上它。

此外，HTTP 1.1 协议添加了内容、编码、字符集，甚至语言的协商机制，还添加了传输编码、缓存指令、客户端 cookie 等十几个可以每次请求都协商的字段。

我们不打算细究 HTTP 1.1 的每个功能，这需要一本书才行，而且已经有很多这方面的好书了。相反，上面的例子只是为了说明 HTTP 的发展有多快，以及客户端与服务器每次交换数据有多复杂。

> 要详细了解 HTTP 协议的内部工作原理，请参考 David Gourley 和 Brian Totty 合著的《HTTP 权威指南》。

9.4　HTTP 2.0：改进传输性能

RFC 2616 自发布以来，一直都是互联网大发展的基石。几十亿大大小小、形形色色的设备，包括桌面计算机和手机，每天都在通过 HTTP 通信传输新闻、视频，还有数以百万计的 Web 应用，它们已经成为我们日常生活不可或缺的部分。

曾经以简单的理念开始，只有一行的用于取得超文本的协议，迅速发展为通用的超媒体传输机制。十几年后的今天，HTTP 已经成为可以在任何领域使用的核心协议。无所不在的支持这个协议的服务器，以及随处可见的访问这些服务器的客户端，都意味着在 HTTP 之上，人们正在设计和部署更多的应用。

需要一个协议来控制你的咖啡壶吗？ RFC 2324 就是“超文本咖啡壶控制协议”（Hyper Text Coffee Pot Control Protocol，HTCPCP/1.0），它是 IETF 愚人节的一个笑话。但在全新的物联网时代，这种事将会越来越多地变为现实。

> HTTP（Hypertext Transfer Protocol）是一个应用层协议，可用于分布协作式的超媒体系统。它是一个通用、无状态的协议。除了超文本，通过扩展它的请求方式、错误编码及首部，还可以将它用于很多其他领域，比如域名服务器和分布式对象管理系统。HTTP 的一个功能就是允许数据的类型变化和协商，从而允许系统独立于被传输的数据构建。
>
> ——RFC 2616：HTTP/1.1（1999 年 6 月）

HTTP 的简单本质是它最初得以采用和后来快速发展的关键。事实上，很多嵌入式设备，比如传感器、致动器，甚至咖啡壶，都在使用 HTTP 作为主要的控制和数据协议。然而，在巨大成功的压力之下，同时随着我们越来越多地把应用（社交媒体、电子邮件、新闻和视频，以及个人的生活与工作内容）部署到 Web 上，HTTP 的问题也出现了。今天，用户和 Web 开发者都迫切想要通过 HTTP 1.1 达到一种几近实时的响应速度和协议性能，而要满足这个需求，仅靠在原协议基础上修修补补是不够的。

为应对这些新挑战，HTTP 必须继续发展。HTTP 工作组已经在 2012 年宣布要开发 HTTP 2.0：

> 当前，出现了一种保持 HTTP 语义，但脱离 HTTP/1.x 消息分帧及语法的协议用法。这种用法被证明有碍于性能，并且是在鼓励滥用底层传输协议。
>
> 本工作组将制定一个新规范，从有序、半双工流的角度重新表达当前 HTTP 的语义。与 HTTP/1.x 一样，主要将使用 TCP 作为传输层，不过也应该支持其他传输协议。
>
> ——HTTP 2.0 纲领（2012 年 1 月）

HTTP 2.0 的主要目标是改进传输性能，实现低延迟和高吞吐量。主版本号的增加听起来像是要做大的改进，从性能角度说的确如此。但从另一方面看，HTTP 的高层协议语义并不会因为这次版本升级而受影响。所有 HTTP 首部、值，以及它们的使用场景都不会变。

现有的任何网站和应用，无需做任何修改都可以在 HTTP 2.0 上跑起来。换句话说，不用为了利用 HTTP 2.0 的好处而修改标记。HTTP 服务器必须运行 HTTP 2.0 协议，但大部分用户都不会因此而受到影响。唯一的不同——假如 HTTP 工作组的目标能够达成，就应该是我们的应用能够以更低的延迟和更高的网络连接利用率交付！

明白了这一点，我们就不会盲目冒进了。在了解 HTTP 2.0 协议的新功能之前，还是有必要先了解一下在当前 HTTP 1.1 基础上部署和提升性能的最佳实践。HTTP 2.0 工作组的进展很快，但即便最终标准已经完成，在可见的未来，我们还是必须支持 HTTP 1.1 客户端——现实一点说，至少十年内还要支持。

第 10 章

Web性能要点

在任何复杂的系统中，性能优化的很大一部分工作就是把不同层之间的交互过程分解开来，弄清楚每一层次交互的约束和限制。到目前为止，我们已经比较详细地分析了一些个别网络组件（不同的物理交付方式和传输协议）。现在，我们把目光转向更宏观的 Web 性能优化：

- 延迟和带宽对 Web 性能的影响；
- 传输协议（TCP）对 HTTP 的限制；
- HTTP 协议自身的功能和缺陷；
- Web 应用的发展趋势及性能需求；
- 浏览器局限性和优化思路。

优化不同层之间的交互与解一组方程没有什么不同，因为不同层之间总是相互依赖，但优化方式却有很多可能性。任何优化建议和最佳做法都不是一成不变的，涉及的每个要素都是动态发展的：浏览器越来越快、用户上网条件不断改善、Web 应用的功能和复杂度也与日俱增。

因此，在讨论具体的最佳实践之前，一定要先明确真正的问题何在：什么是现代 Web 应用，我们手里有什么工具，如何测量 Web 性能，系统的哪些部分对优化有利或有碍？

10.1 超文本、网页和Web应用

互联网在过去几十年的发展过程中，至少带给了我们三种体验：超文本文档、富媒体网页和交互式 Web 应用。不可否认，后两种之间的界限有时候对用户来说很模糊，但从性能的角度看，这几种形式都要求我们在讨论问题、测量和定义性能时采取不同的手段。

- 超文本文档

  万维网就起源于超文本文档，一种只有基本格式，但支持超链接的纯文本文档。按照现代眼光来看，这种文档或许没什么值得大惊小怪的，但它验证了万维网的假设、前景及巨大的实用价值。

- 富媒体网页

  HTML 工作组和早期的浏览器开发商扩展了超文本，使其支持更多的媒体，如图片和音频，同时也为丰富布局增加了很多手段。网页时代到来了，我们可以基于不同的媒体构建可见的页面布局了。但网页还只是看起来漂亮，很大程度上没有交互功能，与可打印的页面没有区别。

- Web应用

  JavaScript 及后来 DHTML 和 Ajax 的加入，再一次革命了 Web，把简单的网页转换成了交互式 Web 应用。Web 应用可以在浏览器中直接响应用户操作。于是，Outlook Web Access（IE5 中的 `XMLHTTP` 就诞生于这个应用）等最早的、成熟的浏览器应用出现，也揭开了脚本、样式表和标记文档之间复杂依赖的新时代。

HTTP 0.9 会话由一个文档请求构成，这对于取得超文本内容完全够用了：一个文档、一个 TCP 连接，然后关闭连接。因此，提升性能就是围绕短期 TCP 连接优化一次 HTTP 请求。

富媒体网页的出现改变了这个局面，因为一个简单的文档，变成了文档加依赖资源。因此，HTTP 1.0 引入了 HTTP 元数据的表示法（首部），HTTP 1.1 又加入了各种旨在提升性能的机制，如缓存、持久连接，等等。事实上，多 TCP 连接目前仍然存在，性能的关键指标已经从文档加载时间，变成了页面加载时间，常简写为 PLT（Page Load Time）。

> PLT 的简单定义就是：“浏览器中的加载旋转图标停止旋转的时间。”更技术的定义则是浏览器中的 `onload` 事件，这个事件由浏览器在文档及其所有依赖资源（JavaScript、图片，等等）下载完毕时触发。

最后，Web 应用把网页的简单依赖关系（在标签中使用媒体作为基本内容的补充）转换成了复杂的依赖关系：标记定义结构、样式表定义布局，而脚本构建最终的交互式应用，响应用户输入，并在交互期间创建样式表和标记。

结果，页面加载时间，这个一直以来衡量 Web 性能的事实标准，作为一个性能基础也越来越显得不够了。我们不再是构建网页，而是在构建一个动态、交互的 Web 应用。除了测量每个资源及整个页面的加载时间（PLT），还要回答有关应用的如下几个问题：

- 应用加载过程中的里程碑是什么？
- 用户第一次交互的时机何在？
- 什么交互应该吸引用户参与？
- 每个用户的参与及转化率如何？

性能好坏及优化策略成功与否，与你定义应用的特定基准和条件，并反复测试的效果直接相关。没有什么比得上应用特定的知识和测量，在关系到赢利目标和商业指标的情况下更是如此。

DOM、CSSOM 和 JavaScript

前所说的“脚本、样式表和标记文档之间复杂依赖”到底指什么呢？要回答这个问题，我们得先回顾一下浏览器架构，了解一下解析、布局和脚本如何相互配合在屏幕上绘制出像素来（图 10-1）。

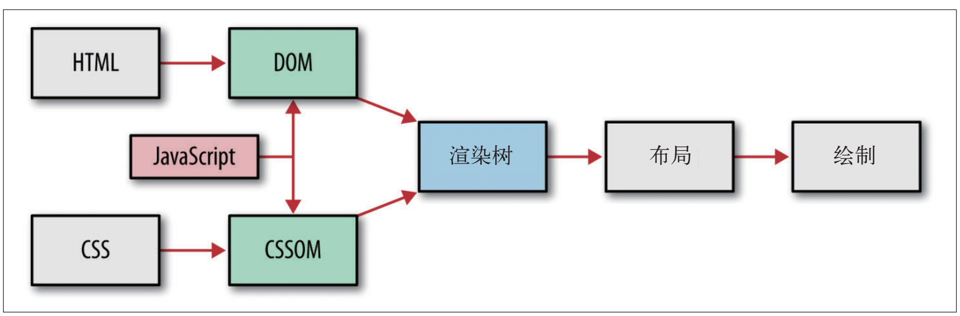


图 10-1：浏览器处理流水线：HTML、CSS 和 JavaScript

浏览器在解析 HTML 文档的基础上构建 DOM（Document Object Model，文档对象模型）。与此同时，还有一个常常被忽略的模型——CSSOM（CSS Object Model，CSS 对象模型），也会基于特定的样式表规则和资源构建而成。这两个模型共同创建“渲染树”，之后浏览器就有了足够的信息去进行布局，并在屏幕上绘制图形。到目前为止，一切都很好理解。

> 然而，此时不得不提到我们最大的朋友和祸害：JavaScript。脚本执行过程中可能遇到一个同步的 `document.write`，从而阻塞 DOM 的解析和构建。类似地，脚本也可能查询任何对象的计算样式，从而阻塞 CSS 处理。结果，DOM 及 CSSOM 的构建频繁地交织在一起：DOM 构建在 JavaScript 执行完毕前无法进行，而 JavaScript 在 CSSOM 构建完成前也无法进行。
>
> 应用的性能，特别是首次加载时的“渲染前时间”，直接取决于标记、样式表和 JavaScript 这三者之间的依赖关系。顺便说一句，还记得流行的“样式在上，脚本在下”的最佳实践吗？现在你该知道为什么了。渲染和脚本执行都会受样式表的阻塞，因此必须让 CSS 以最快的速度下载完。

10.2　剖析现代Web应用

现代 Web 应用到底长啥样？ HTTP Archive（http://httparchive.org/）可以回答这个问题。这个网站项目一直在抓取世界上是热门的网站（Alexa 前 100 万名中的 30 多万名），记录、聚合并分析每个网站使用的资源、内容类型、首部及其他元数据的数量。

2013 年初，一个普通的 Web 应用由下列内容构成。

- 90 个请求，发送到 15 个主机，总下载量 1311 KB
  - HTML：10 个请求，52 KB
  - 图片：55 个请求，812 KB
  - JavaScript：15 个请求，216 KB
  - CSS：5 个请求，36 KB
  - 其他资源：5 个请求，195 KB

在你读到这里的时候，前面所有数字都会变大（图 10-2），持续向上的趋势一直都很稳定，而且没有停止的迹象。不过，抛开实际的请求和字节数不提，更值得关注的还是个别组件的量级：现在，一个普通的 Web 应用大约就有 1 MB，有 100 个左右的资源分散在 15 台不同的主机上！

与桌面应用相比，Web 应用不需要单独安装，只要输入 URL，按下回车键，就可以正常运行。可是，桌面应用只需要安装一次，而 Web 应用每次访问都需要走一遍“安装过程”——下载资源、构建 DOM 和 CSSOM、运行 JavaScript。正因为如此，Web 性能研究迅速发展，成为人们热议的话题也就不足为怪了。上百个资源、成兆字节的数据、数十个不同的主机，所有这些都必须在短短几百 ms 内亲密接触一次，才能带来即刻呈现的 Web 体验。

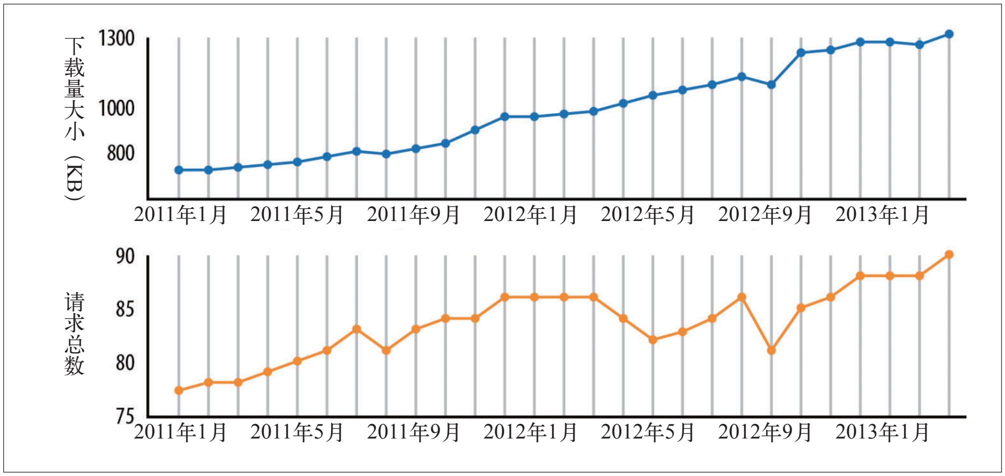


图 10-2：平均规模和请求数量（HTTP Archive）

10.2.1　速度、性能与用户期望

速度和性能是两个相对的概念。每个应用都要满足自己特定的需求，因为商业条件、应用场景、用户期望，以及功能复杂性各不相同。尽管如此，如果应用必须对用户作出响应，那我们就必须从用户角度来考虑可感知的处理时间这个常量。事实上，虽然生活节奏越来越快——至少我们感觉如此，但人类的感知和反应时间则一直都没有变过（表 10-1），而且与应用的类型（在线或离线）、媒体的类型（笔记本、台式机或移动设备）无关。

表10-1：时间和用户感觉

时间	感觉
0 ~100 ms	很快
100~300 ms	有一点点慢
300~1000 ms	机器在工作呢
> 1000 ms	先干点别的吧
> 10000 ms	不能用了

这个表格解释了 Web 性能社区总结的经验法则：必须 250 ms 内渲染页面，或者至少提供视觉反馈，才能保证用户不走开！

如果想让人感觉很快，就必须在几百 ms 内响应用户操作。超过 1 s，用户的预期流程就会中断，心思就会向其他任务转移，而超过 10 s，除非你有反馈，否则用户基本上就会终止任务！

现在，把 DNS 查询，随后的 TCP 握手，以及请求网页所需的几次往返时间都算上，光网络上的延迟就能轻易突破 100~1000 ms 的预算（参见表 8-1）。难怪有那么多用户，特别是那些移动或无线用户，抱怨上网速度慢了！

> Jakob Nielsen 的 *Usability Engineering* 和 Steven Seow 的 *Designing and Engineering Time* 是每一位开发者和设计师都应该好好读的书！时间测量是客观的，而时间感知是主观的。我们可以通过设计来改善感知性能。

把性能变成钞票

速度是一种功能，而非简单的为了速度而速度。谷歌、微软和亚马逊的研究都表明，性能可以直接转换成收入。比如，Bing 搜索网页时延迟 2000 ms 会导致每用户收入减少 4.3%。

类似地，Aberdeen 一项覆盖 160 多家组织的研究表明，页面加载时间增加 1 秒，会导致转化率损失 7%，页面浏览量减少 11%，客户满意度降低 16%！

网络越快，PV 越多，黏性越强，转化率越高。不过，不能只听这些行业调查的结论，你还得根据自己的标准来测量自己网站的性能。至于为什么，请往下看，或者直接跳到 10.4 节“人造和真实用户性能度量”。

10.2.2　分析资源瀑布

谈到 Web 性能，必然要谈资源瀑布。事实上，资源瀑布很可能是我们可以用来分析网络性能，诊断网络问题的一个最有价值的工具。很多浏览器都内置了一些手段，让我们能查看资源瀑布。此外，还有一些不错的在线工具，比如 WebPageTest（http://www.webpagetest.org/），可以在不同的浏览器中呈现资源瀑布。

> WebPageTest.org 是一个开源免费的项目，可以测试世界各地网页的性能。测试用的浏览器在虚拟机中运行，可编程，可配置，有各种连接和浏览器设置可选。测试完成后，通过网页就可以查看结果。这一切使得 WebPageTest 成为无与伦比的 Web 性能测试工具。

首先，必须知道每一个 HTTP 请求都由很多独立的阶段构成（图 10-3）：DNS 解析、TCP 连接握手、TLS 协商（必要时）、发送 HTTP 请求，然后下载内容。这些阶段的时长在不同的浏览器中会略有不同，但为了简单起见，本章就使用 WebPageTest 测试的结果。请大家提前熟悉每种颜色在自己浏览器中的含义。

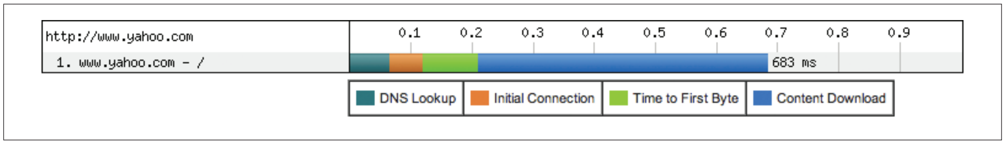


图 10-3：HTTP 请求的构成（WebPageTest）

仔细看一下图 10-3，打开 Yahoo! 主页花了 683 ms，而其中有 200 多 ms 在等待网络就绪，占到了请求延迟的 30%！然而，请求文档还只是开始，现代的 Web 应用还需要各种资源（图 10-4）配合来生成最终结果。准确地说，要加载 Yahoo! 主页，浏览器要请求 52 个资源，从 30 个不同的主机获得，这些资源加起来总共 486 KB。

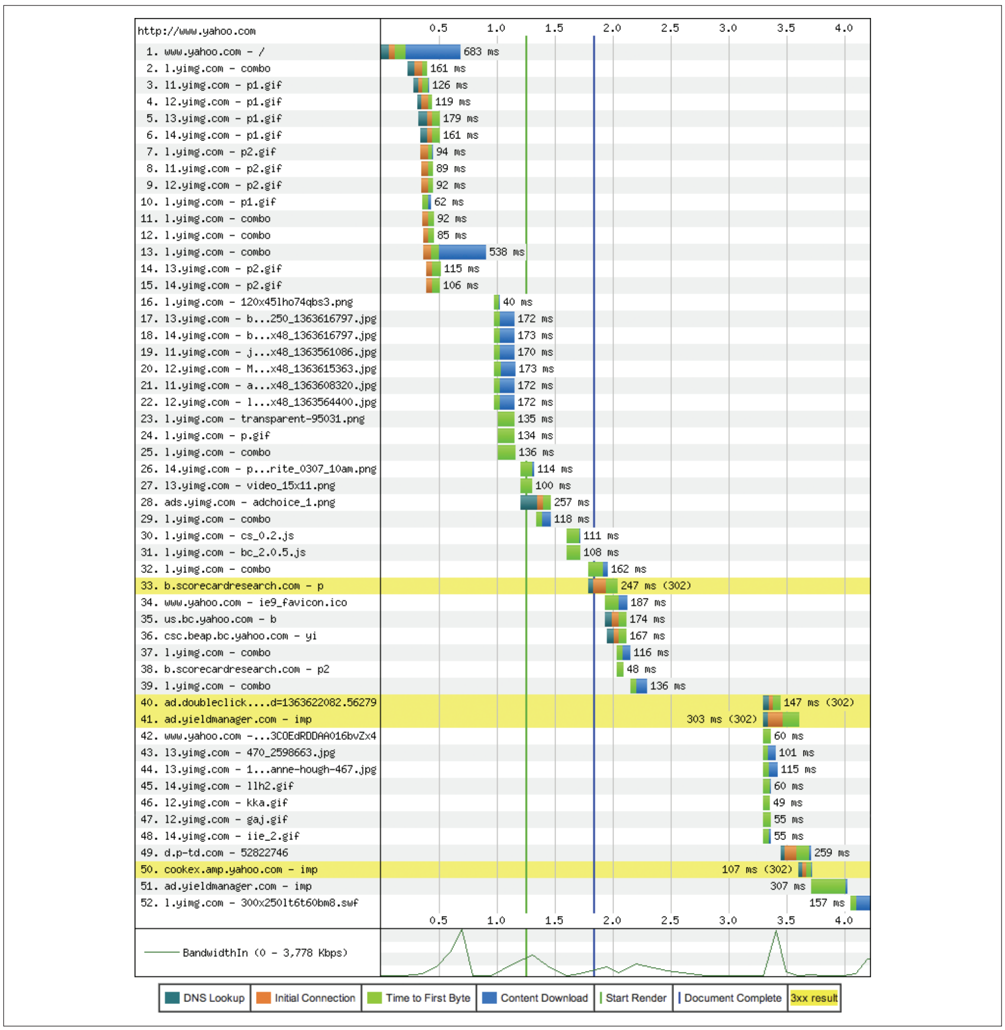


图 10-4：Yahoo.com 资源瀑布图（WebPageTest，2013 年 3 月）

资源的瀑布图能够揭示出页面的结构和浏览器处理顺序。首先，取得 www.yahoo.com 对应的文档，同时分派新的 HTTP 请求：HTTP 解析是递增执行的，这样浏览器可以及早发现必要的资源，然后并行发送请求。实际上，何时获得什么资源很大程度上取决于标记结构。浏览器可以变更某些请求的优先顺序，但递增地发现文档中的每一个资源，最终造就了不同资源间的“瀑布效果”。

其次，“Start Render”（绿色的竖线）会在所有资源下载完成前开始，以便用户在页面构建期间就能与之交互。其实，“Document Complete”事件（蓝色的竖线）也会在剩余资源下载完成前触发。换句话说，浏览器的加载旋转图标此时停止旋转，用户可以继续与页面交互，但 Yahoo! 主页会渐进地在后台填充后续内容，比如广告和社交部件。

最早渲染时间、文档完成时间和最后资源获取时间，这三个时间说明我们讨论 Web 性能时有三个不同测量指标。我们应该关注哪一个时间呢？答案并不唯一，因应用而不同！ Yahoo! 的工程师们选择了利用浏览器递增加载机制，让用户能够尽早与重要内容交互。而这样一来，他们必须根据应用不同，确定哪些内容重要（须先加载），哪些内容不重要（可以后填充）。

> 什么时候请求什么资源，以什么顺序请求资源，这在不同的浏览器中可能有不同的实现。结果，你的应用性能也会因浏览器而异。
> 提示：WebPageTest 支持选择位置和浏览器。

网络的瀑布图是个很强大的工具，有助于揭示任何页面或应用是否处于优化状态。前面分析和优化资源瀑布的过程，一般称为*前端性能分析和优化*。不过，这个称呼有误导性，好像所有性能瓶颈都在客户端似的。实际上，尽管 JavaScript、CSS 和渲染流水线很重要，而且资源对性能影响很大，但服务器响应时间和网络延迟（“后端性能”）对资源瀑布的影响也不容忽视。毕竟，如果网络被阻塞，也就谈不上什么解析或运行资源了！

为了说明这一点，只要切换到 WebPageTest 资源瀑布的连接视图即可（图 10-5）。

资源瀑布图记录的是 HTTP 请求，而连接视图展示了每个 TCP 连接（这里共 30 个）的生命期，这些连接用于获取 Yahoo! 主页的资源。哪里比较突出呢？注意蓝色的下载时间，很短，在每个连接的总延迟里几乎微不足道。这里总共发生了 15 次 DNS 查询，30 次 TCP 握手，还有很多等待接收每个响应第一个字节的网络延迟（绿色）。

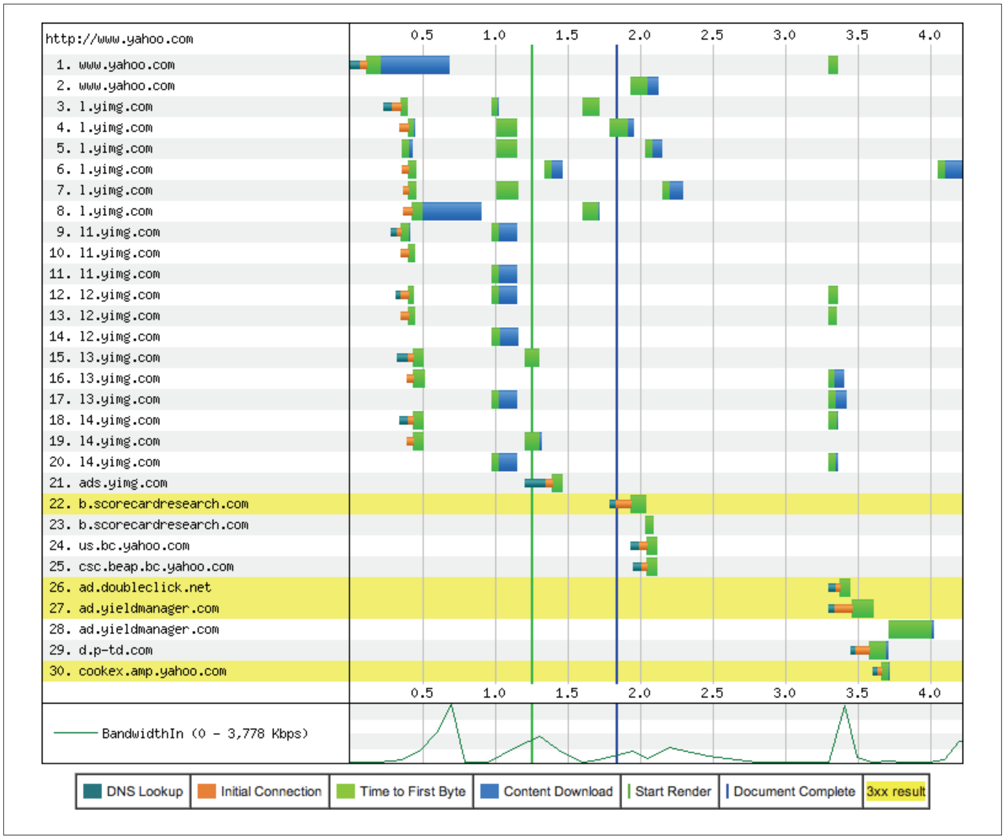


图 10-5：Yahoo.com 资源瀑布图的连接视图（WebPageTest，2013 年 3 月）

> 为什么有些请求只显示绿色条（第一字节接收时间）？因为很多响应很小，相应的下载时间没有在图中体现。事实上，请求、响应的主要时间通常是往返延迟和服务器处理时间。

最后，也是最重要的，连接视图的底部显示了带宽利用率曲线。除了少量数据爆发外，可用连接的带宽利用率很低。这说明性能的限制并不在带宽！难道这是个异常现象，或者浏览器问题？都不是。对大多数应用而言，带宽的的确确不是性能的限制因素。限制 Web 性能的主要因素是客户端与服务器之间的网络往返延迟。

10.3　性能来源：计算、渲染和网络访问

Web 应用的执行主要涉及三个任务：取得资源、页面布局和渲染、JavaScript 执行。其中，渲染和脚本执行在一个线程上交错进行，不可能并发修改生成的 DOM。实

际上，优化运行时的渲染和脚本执行是至关重要的，可以参考 10.1 节中的“DOM、CSSOM 和 JavaScript”。

可是，就算优化了 JavaScript 执行和渲染管道，如果浏览器因网络阻塞而等待资源到来，那结果也好不到哪里去。对运行在浏览器中的应用来说，迅速而有效地获取网络资源是第一要义。

有人可能会问，互联网今天的速度不是快多了吗，难道网络还会阻塞？是的，我们的应用也比以前大多了。然而，假如真像每个 ISP 和移动运营商鼓吹的那样，全球平均网速已经达到 3.1 Mbit/s（参见 1.6 节“网络边缘的带宽”），而且还在继续提速，Web 应用大一点又算得了什么呢？可惜的是，凭直觉以及前面展示的 Yahoo! 的例子，我们知道如果真是这样，那你就不用再看这本书了。好，下面我们仔细谈一谈。

> 要想详细了解带宽和延迟的发展趋势及其相互影响，请参考第 1 章“延迟与带宽”。

10.3.1　更多带宽其实不（太）重要

先别急，带宽当然重要！毕竟，所有 ISP 和移动运营商的广告，都意在提醒我们高带宽的好处：上传和下载加速、更流畅地欣赏视频，都有赖于 [*请读者在此插入最新数字*] Mbit/s 的速度！

能接入更高带宽固然好，特别是传输大块数据时更是如此，比如在线听音乐、看视频，或者下载大文件。可是，日常上网浏览需要的是从数十台主机获取较小的资源，这时候往返时间就成了瓶颈：

- 在 Yahoo! 主页上看视频受限于带宽；
- 加载和渲染 Yahoo! 主页受限于延迟。

根据在线视频品质及编码不同，需要的带宽从几百 Kbit/s 到几 Mbit/s 不等。比如，要流畅观看 1080P 的高清视频，需要 3 Mbit/s 以上的带宽。这个带宽是很多用户触手可及的，Netflix 等在线视频网站的流行也印证了这一点。那么，为什么下载比视频小得多得多 Web 应用，在能流畅观看高清视频的连接上却成了难题呢？

10.3.2　延迟是性能瓶颈

关于为什么延迟会成为浏览网页的限制因素，前几章已经介绍了足够多的理论知识，只是那些都只能给我们一个定性的认识。下面，我们就来通过两张图（图 10-6），定

量地感受一下不同的带宽和延迟时间对页面加载时间分别有什么影响。这两张图来自 SPDY 协议的开发者之一 Mike Belshe，测试的是互联网上最热门的一些站点。

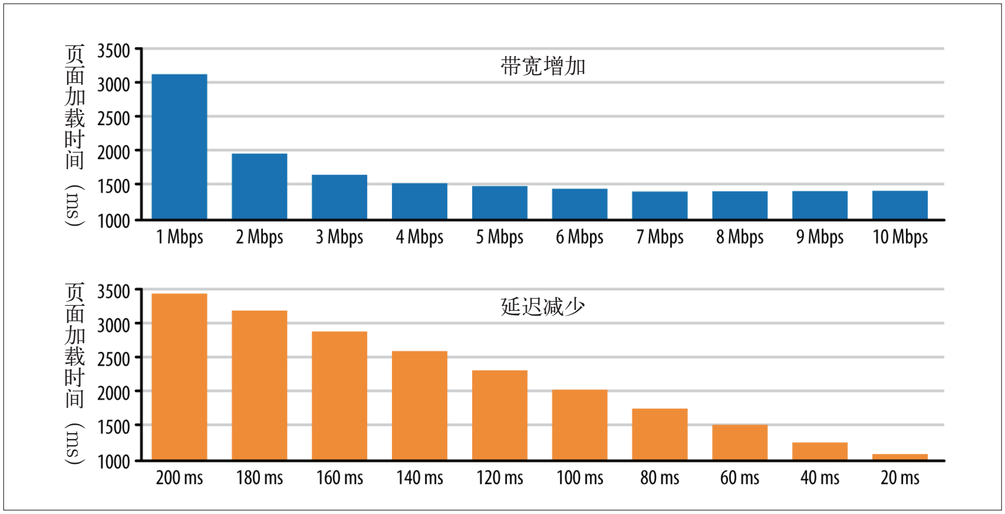


图 10-6：页面加载时间与带宽和延迟的关系

> Mike Belshe 的研究是谷歌开发 SPDY 协议的依据，而 SPDY 就是后来 HTTP 2.0 的基础。

在第一个测试中，连接的延迟时间固定而带宽递增，从 1 Mbit/s 依次递增至 10 Mbit/s。注意一开始，从 1 Mbit/s 升级到 2 Mbit/s，页面加载时间几乎减少了一半，这也是我们希望看到的结果。可是，在此之后，带宽递增，加载时间减少得越来越不明显。而当带宽超过 5 Mbit/s 时，加载时间的减少比例只有几个百分点，从 5 Mbit/s 升级到 10 Mbit/s，页面加载时间仅降低了 5%！

Akamai 公司的宽带速度报告（参见 1.6 节“网络边缘的带宽”）显示，美国普通消费者上网带宽已经达到 5 Mbit/s 以上。很多其他国家很快也能达到这个数字，甚至有的已经超过了它。由此可知，靠提高带宽不会给美国人浏览网页带来多大的性能提升。或许美国人下载大文件、看视频的速度很快，但加载包含这些文件的页面的时间不会明显缩短。换句话说，*增加带宽没有那么重要*。

然而，在延迟以 20 ms 递减的试验中，页面加载时间呈线性减少趋势。在选择 ISP 时，是不是应该把延迟时间而不是带宽放在首位呢？

> 要让互联网从整体上提速，就必须寻求降低 RTT 的方法。把跨大西洋的 RTT 从 150 ms 降低到 100 ms，结果会怎么样？结果比把用户带宽从 3.9 Mbit/s 提高到 10 Mbit/s 甚至 1 Gbit/s 对速度的影响都大。
>
> 减少页面加载时间的另一种方法，就是减少加载每个页面过程中的往返次数。今天，加载每个页面都需要客户端到服务器之间的数次往返。这些往返很大程度上是由客户端与服务器之间为建立连接（DNS、TCP、HTTP 等）而进行握手所导致的，当然有的是由通信协议引发的（如 TCP 慢启动）。假如我们能够对协议加以改进，使得加载同样多的数据只需更少的往返，那应该也能够减少页面加载时间。这就是 SPDY 的目标之一。
>
> ——Mike Belshe，更多带宽其实不（太）重要

很多人可能都会惊讶于这两项试验的结果，而实际情况的确如此，这正是 TCP 握手机制、流量和拥塞控制、由丢包导致的队首拥塞等底层协议特点影响性能的直接后果。大多数 HTTP 数据流都是小型突发性数据流，而 TCP 则是为持久连接和大块数据传输而进行过优化的。网络往返时间在大多数情况下都是 TCP 吞吐量和性能的限制因素，详细信息可参见 2.5 节“针对 TCP 的优化建议”。于是，延迟自然也就成了 HTTP 及大多数基于 HTTP 交付的应用的性能瓶颈。

> 如果延迟对大多数有线连接是限制性能的因素，那可想而知，它对无线客户端将是更重要的性能瓶颈。事实上，无线延迟明显更高，因此网络优化是提升移动 Web 应用性能的关键。

10.4　人造和真实用户性能度量

有度量，才有改进。问题在于度量的标准是否正确，过程是否合理。如前所述，度量现代 Web 应用的性能并不容易，没有什么指标放之四海皆准。换句话说，我们必须定义自己的指标。确定了度量标准之后，接下来要收集性能数据，这个过程涉及一系列人造和真实用户的性能度量。

宽泛地说，在受控度量环境下完成的任何测试都可称为人造测试。首先，本地构建过程运行性能套件，针对基础设施加载测试，或者针对一组分散在各地的监控服务器加载测试，这些服务器定时运行脚本并记录输出。这些测试中的任何一个都可能测试不同的基础设施（如应用服务器的吞吐量、数据库性能、DNS 时间，等等），并作为稳定的基准辅助检测性能衰退或聚焦于系统的某个特定组件。

> 只要配置得当，人造测试就可以提供一个受控且可重现的性能测试环境。而这个环境非常适合发现和修复性能问题，确保用户体验。提示：确定一个关键的性能指标，并为它们分别设定一个“预算额度”，纳入人造测试计划。一旦哪个指标超出“预算”，马上拉响警报！

可是，人造测试不能发现所有性能瓶颈。特别是在人造环境下收集到的性能数据缺乏现实当中的多样性，难以据之确定应用带给用户的最终体验。这个问题有如下表现。

- 场景及页面选择：很难重复真实用户的导航模式；
- 浏览器缓存：用户缓存不同，性能差别很大；
- 中介设施：中间代理和缓存对性能影响很大；
- 硬件多样化：不同的 CPU、GPU 和内存比比皆是；
- 浏览器多样化：各种浏览器版本，有新有旧；
- 上网方式：真实连接的带宽和延迟可能不断变化。

上述这些方面，加之其他一些类似情况，意味着除了人造测试，我们必须通过真实用户度量（RUM，Real-User Measurement）来获取用户使用我们应用的真实性能数据，从而确保性能度量的有效性。有一个好消息，W3C Web Performance Working Group 通过引入 Navigation Timing API（图 10-7）为我们做真实用户测试提供了便利，这个 API 目前已得到很多现代桌面和移动浏览器的支持。

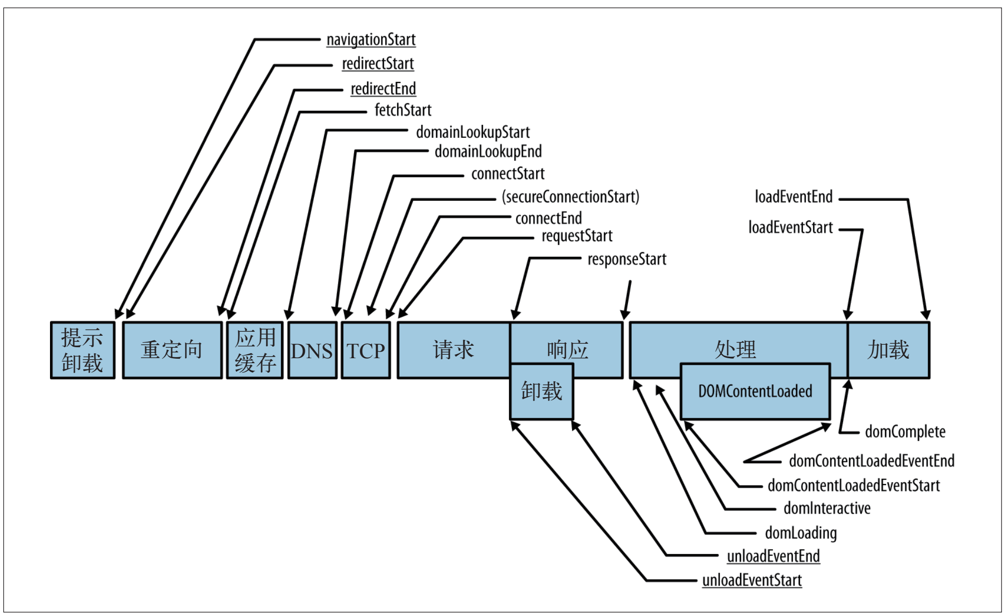


图 10-7：Navigation Timing 监测到的特定于用户的计时器

> 2014 年初，支持 Navigation Timing 的浏览器有 IE9+、Chrome 6+、Firefox 7+ 和 Opera 15+，包括桌面和移动版本，Safari 还不支持。要了解最新情况，请参考 http://caniuse.com/nav-timing。

Navigation Timing 的真正好处是它提供了以前无法访问的数据，比如 DNS 和 TCP 连接时间，而且精确度极高（微秒级时间戳）。要获得这些数据，可以在浏览器中访问标准的 `performance.timing` 对象。实际上，收集这些数据的过程很简单：加载页面，从用户浏览器中取得相应的计时对象，然后将其传回分析服务器！通过观察这些数据，就可以知道用户使用我们应用时的真实性能，发现不同硬件和不同网络连接导致的差异。

分析真实用户度量数据

分析性能数据时，要时时关注数据的潜在分布，抛开均值，多用直方图、中位值和分位数。对于偏态分布和多重模态分布，均值是个没有意义的指标。图 10-8 展示了同一站点度量数据的这两种分布，其中偏态分布显示的是页面加载时间，而多重模态分布显示的是服务器响应时间（两种模式分别是应用服务器生成有缓存和无缓存页面的时间）。

页面加载时间段（秒）	页面加载采样	百分比
0 - 1	22	5.35%
1 - 3	116	28.22%
3 - 7	148	36.01%
7 - 13	66	16.06%
13 - 21	22	5.35%
21 - 35	14	3.41%
35 - 60	10	2.43%

服务器响应时间段（秒）	响应采样	百分比
0 - 0.01	18	4.40%
0.01 - 0.10	33	8.07%
0.10 - 0.50	168	41.08%
0.50 - 1	22	5.38%
1 - 2	124	30.32%
2 - 5	38	9.29%
5+	6	1.47%

图 10-8：igvita.com 的页面加载时间（偏态）和响应时间（多重模态）分布

要确保自己的分析工具能够针对性能数据得到正确的统计结果。前面的数据来自 Google Analytics，它在标准的“站点速度”报告中提供了直方图。Google Analytics 会在安装分析跟踪器的情况下自动收集 Navigation Timing 数据。类似地，还有很多第三方开发商提供了基于 Navigation Timing 的数据收集和报告工具。

最后，除了 Navigation Timing，W3C Performance Group 还标准化了另外两个 API：User Timing 和 Resource Timing。Navigation Timing 只针对根文档提供性能计时器，Resource Timing 则针对页面中的每个资源都提供类似的性能数据，可以让我们收集到关于页面的完整性能数据。类似地，User Timing 也是一个简单的 JavaScript API，可以标记和度量特定应用的性能指标，提供高精确度的计时结果：

```
function init() {
  performance.mark("startTask1"); ❶
  applicationCode1(); ❷
  performance.mark("endTask1");

  logPerformance();
}

function logPerformance() {
  var perfEntries = performance.getEntriesByType("mark");
  for (var i = 0; i < perfEntries.length; i++) { ❸
    console.log("Name: " + perfEntries[i].name +
                " Entry Type: " + perfEntries[i].entryType +
                " Start Time: " + perfEntries[i].startTime +
                " Duration: "   + perfEntries[i].duration  + "\n");
  }
  console.log(performance.timing); ❹
}
```

❶ 存储（标记）时间戳，并命名（startTask1）

❷ 执行应用代码

❸ 迭代和记录用户计时数据

❹ 记录当前页面的 Navigation Timing 对象

综合运用 Navigation、Resource 和 User 计时 API，可以对每个 Web 应用的真实用户性能进行度量，再也不要说没有精确的数据了！优化要以度量为依据，RUM 和人造测试是互为补充的手段，可以帮我们发现回归现象和真正的瓶颈，提升应用的用户体验。

> 可靠的性能优化策略源自特定于应用的自定义指标。不存在唯一度量和定义用户体验的方式。相反，我们必须针对每个应用定义和设计特定的里程碑和活动，这是一个涉及所有干系人（公司老总、设计师和开发人员）的协作过程。

10.5　针对浏览器的优化建议

不得不说，浏览器可远远不止一个网络套接字管理器那么简单。性能可以说是每个浏览器开发商的核心卖点，既然性能如此重要，那浏览器越来越聪明也就毫不奇怪

了。预解析可能的 DNS 查询、预连接可能的目标、预取得和优先取得重要资源，这些都是浏览器变聪明的标志。

可行的优化手段会因浏览器而异，但从核心优化策略来说，可以宽泛地分为两类。

- 基于文档的优化
  熟悉网络协议，了解文档、CSS 和 JavaScript 解析管道，发现和优先安排关键网络资源，尽早分派请求并取得页面，使其尽快达到可交互的状态。主要方法是优先获取资源、提前解析等。

- 推测性优化
  浏览器可以学习用户的导航模式，执行推测性优化，尝试预测用户的下一次操作。然后，预先解析 DNS、预先连接可能的目标。

好消息是，所有这些优化都由浏览器替我们自动完成，经常可以节省几百 ms 的网络延迟。既然如此，那理解这些优化背后的原理就至关重要了，这样才能利用浏览器的这些特性，提升应用性能。大多数浏览器都利用了如下四种技术。

- 资源预取和排定优先次序
  文档、CSS 和 JavaScript 解析器可以与网络协议层沟通，声明每种资源的优先级：初始渲染必需的阻塞资源具有最高优先级，而低优先级的请求可能会被临时保存在队列中。

- DNS预解析
  对可能的域名进行提前解析，避免将来 HTTP 请求时的 DNS 延迟。预解析可以通过学习导航历史、用户的鼠标悬停，或其他页面信号来触发。

- TCP预连接
  DNS 解析之后，浏览器可以根据预测的 HTTP 请求，推测性地打开 TCP 连接。如果猜对的话，则可以节省一次完整的往返（TCP 握手）时间。

- 页面预渲染
  某些浏览器可以让我们提示下一个可能的目标，从而在隐藏的标签页中预先渲染整个页面。这样，当用户真的触发导航时，就能立即切换过来。

> 要了解谷歌 Chrome 浏览器是如何实现这些及其他网络优化机制的，请参考“High Performance Networking in Google Chrome”（http://t.cn/zYy9ni6）。

从外部看，现代浏览器的网络协议实现以简单的资源获取机制的面目示人，而从内部来说，它又极为复杂精密，为了解如何优化性能，非常值得深入钻研。那么，在

探寻的过程中，我们怎么利用浏览器的这些机制呢？首先，要密切关注每个页面的结构和交付：

- CSS 和 JavaScript 等重要资源应该尽早在文档中出现；
- 应该尽早交付 CSS，从而解除渲染阻塞并让 JavaScript 执行；
- 非关键性 JavaScript 应该推迟，以避免阻塞 DOM 和 CSSOM 构建；
- HTML 文档由解析器递增解析，从而保证文档可以间隙性发送，以求得最佳性能。

除了优化页面结构，还可以在文档中嵌入提示，以触发浏览器为我们采用其他优化机制：

```
<link rel="dns-prefetch" href="//hostname_to_resolve.com"> ❶
<link rel="subresource"  href="/javascript/myapp.js"> ❷
<link rel="prefetch"     href="/images/big.jpeg"> ❸
<link rel="prerender"    href="//example.org/next_page.html"> ❹
```

❶ 预解析特定的域名
❷ 预取得页面后面要用到的关键性资源
❸ 预取得将来导航要用的资源
❹ 根据对用户下一个目标的预测，预渲染特定页面

这里的每一个提示都会触发一个推测性优化机制。浏览器虽然不能保证落实，但可以利用这些提示优化加载策略。可惜的是，并非所有浏览器都支持这些提示（表 10-2）。不过，如果它们不支持，也只会把提示当成空操作，有益无害。因此，一定要尽可能利用这些手段。

表10-2：推测性浏览器优化提示

浏览器	DNS预获取	重要资源	预获取	预渲染
Firefox	3.5+	N/A	3.5+	N/A
Chrome	1.0+	1.0+	1.0+	13+
Safari	5.01+	N/A	N/A	N/A
IE	9+（预获取）	N/A	10+	11+

IE9 支持“DNS 预获取”，但叫“预获取”（prefetch）。在 IE10+ 中，dns-prefetch 和 prefetch 是等价的，都能用于指定 DNS 预获取。

对大多数用户甚至 Web 开发者而言，DNS、TCP 和 SSL 延迟完全不可见，它们都是在网络层协商确定的，我们中很少有人关注它们。然而，这其中每一步都关乎整体的用户体验，因为每一次额外的网络往返都会增加几十甚至几百 ms 的网络延迟。通过帮助浏览器预测这些往返，可以消除这些瓶颈，从而向用户交付更快更好的体验。

谷歌搜索对 TTFB（Time To First Byte）的优化

HTML 文档在浏览器中是递增解析的。什么意思呢？就是服务器会尽可能频繁发送文档的每一可用部分。这样客户端才能尽早发现和获取关键资源。

谷歌搜索是利用这一技术的最佳范例。接到搜索请求后，服务器立即向浏览器发送搜索页面的静态头部（或页眉），此时甚至尚未分析查询。也是啊，为什么要耽搁呢，每个搜索页面的头部都是一样的！然后，在客户端解析头部的同时，服务器开始搜索索引，在结果准备好之后，再将包含搜索结果的文档剩余部分发送给客户端。此时，再通过 JavaScript 将头部的动态内容（比如登录用户的名字）写进去。

第 11 章

HTTP 1.x

HTTP 1.0 的优化策略非常简单，就一句话：升级到 HTTP 1.1。完了！

改进 HTTP 的性能是 HTTP 1.1 工作组的一个重要目标，后来这个版本也引入了大量增强性能的重要特性，其中一些大家比较熟知的有：

- 持久化连接以支持连接重用；
- 分块传输编码以支持流式响应；
- 请求管道以支持并行请求处理；
- 字节服务以支持基于范围的资源请求；
- 改进的更好的缓存机制。

当然，这些只是其中一部分，要全面讨论 HTTP 1.1 的所有增强特性，非得用一本书不可。同样，推荐大家买一本《HTTP 权威指南》（David Gourley 和 Brian Totty 合著）放在手边。另外，提到好的参考书，Steve Souder 的《高性能网站建设指南》中概括了 14 条规则，有一半针对网络优化：

- *减少DNS查询*

  每次域名解析都需要一次网络往返，增加请求的延迟，在查询期间会阻塞请求。

- *减少HTTP请求*

  任何请求都不如没有请求更快，因此要去掉页面上没有必要的资源。

- 使用CDN

  从地理上把数据放到接近客户端的地方，可以显著减少每次 TCP 连接的网络延迟，增加吞吐量。

- 添加Expires首部并配置ETag标签

  相关资源应该缓存，以避免重复请求每个页面中相同的资源。Expires 首部可用于指定缓存时间，在这个时间内可以直接从缓存取得资源，完全避免 HTTP 请求。ETag 及 Last-Modified 首部提供了一个与缓存相关的机制，相当于最后一次更新的指纹或时间戳。

- Gzip资源

  所有文本资源都应该使用 Gzip 压缩，然后再在客户端与服务器间传输。一般来说，Gzip 可以减少 60%~80% 的文件大小，也是一个相对简单（只要在服务器上配置一个选项），但优化效果较好的举措。

- 避免HTTP重定向

  HTTP 重定向极其耗时，特别是把客户端定向到一个完全不同的域名的情况下，还会导致额外的 DNS 查询、TCP 连接延迟，等等。

上面每一条建议都经受了时间检验，无论是该书出版的 2007 年还是今天，都是适用的。这并不是巧合，而是因为所有这些建议都反映了两个根本方面：消除和减少不必要的网络延迟，把传输的字节数降到最少。这两个根本问题永远是优化的核心，对任何应用都有效。

可是，对所有 HTTP 1.1 的特性和最佳实践，我们就不能这么说了。因为有些 HTTP 1.1 特性，比如请求管道，由于缺乏支持而流产，而其他协议限制，比如队首响应阻塞，则导致了更多问题。为此，Web 开发社区（一直都最有创造性），创造和推行了很多自造的优化手段：域名分区、连接文件、拼合图标、嵌入代码，等等，不下数十种。

对多数 Web 开发者而言，所有这些都是切实可行的优化手段：熟悉、必要，而且通用。可是，现实当中，我们应该对这些技术有正确的认识：它们都是些针对当前 HTTP 1.1 协议的局限性而采用的权宜之计。我们本来不应该操心去连接文件、拼合图标、分割域名或嵌入资源。但遗憾的是，“不应该”并不是务实的态度：这些优化手段之所以存在，都是有原因的，在背后的问题被 HTTP 的下一个版本解决之前，必须得依靠它们。

让 iTunes 用户感受到 3 倍以上的性能增强

在 WWDC 2012 上，Joshua Graessley 分享了一个针对 HTTP 优化取得巨大成效的案例：苹果工程师通过使用 HTTP 的持久连接和管道，重用 iTunes 中既有的 TCP 连接，使得低网速用户的性能提升到原来的 3 倍！

很明显，理解 HTTP 的基本原理回报巨大。要了解前面案例的详细分析，请查找 WWDC 关于 iTunes 的存档，找到 Graessley 的演讲，即 Session 706: Networking Best Practices。

11.1 持久连接的优点

HTTP 1.1 的一个主要改进就是引入了持久 HTTP 连接 。在 9.3 节“HTTP 1.1：互联网标准”中，我们介绍过持久连接，现在我们再演示一下为什么这个特性对我们的优化策略如此重要。

为简单起见，我们限定最多只有一个 TCP 连接，并且只取得两个小文件（每个 <4 KB）：一个 HTML 文档，一个 CSS 文件，服务器响应需要不同的时间（分别为 40 ms 和 20 ms）。

> 图 11-1 假设从纽约到伦敦的单向光纤延迟与表 1-1 中相同，都是 28 ms。

每个 TCP 连接开始都有三次握手，要经历一次客户端与服务器间完整的往返。此后，会因为 HTTP 请求和响应的两次通信而至少引发另一次往返。最后，还要加上服务器处理时间，才能得到每次请求的总时间。

服务器处理时间无法预测，因为这个时间因资源和后端硬件而异。不过，这里的重点其实是由一个新 TCP 连接发送的 HTTP 请求所花的总时间，最少等于两次网络往返的时间：一次用于握手，一次用于请求和响应。这是所有非持久 HTTP 会话都要付出的固定时间成本。

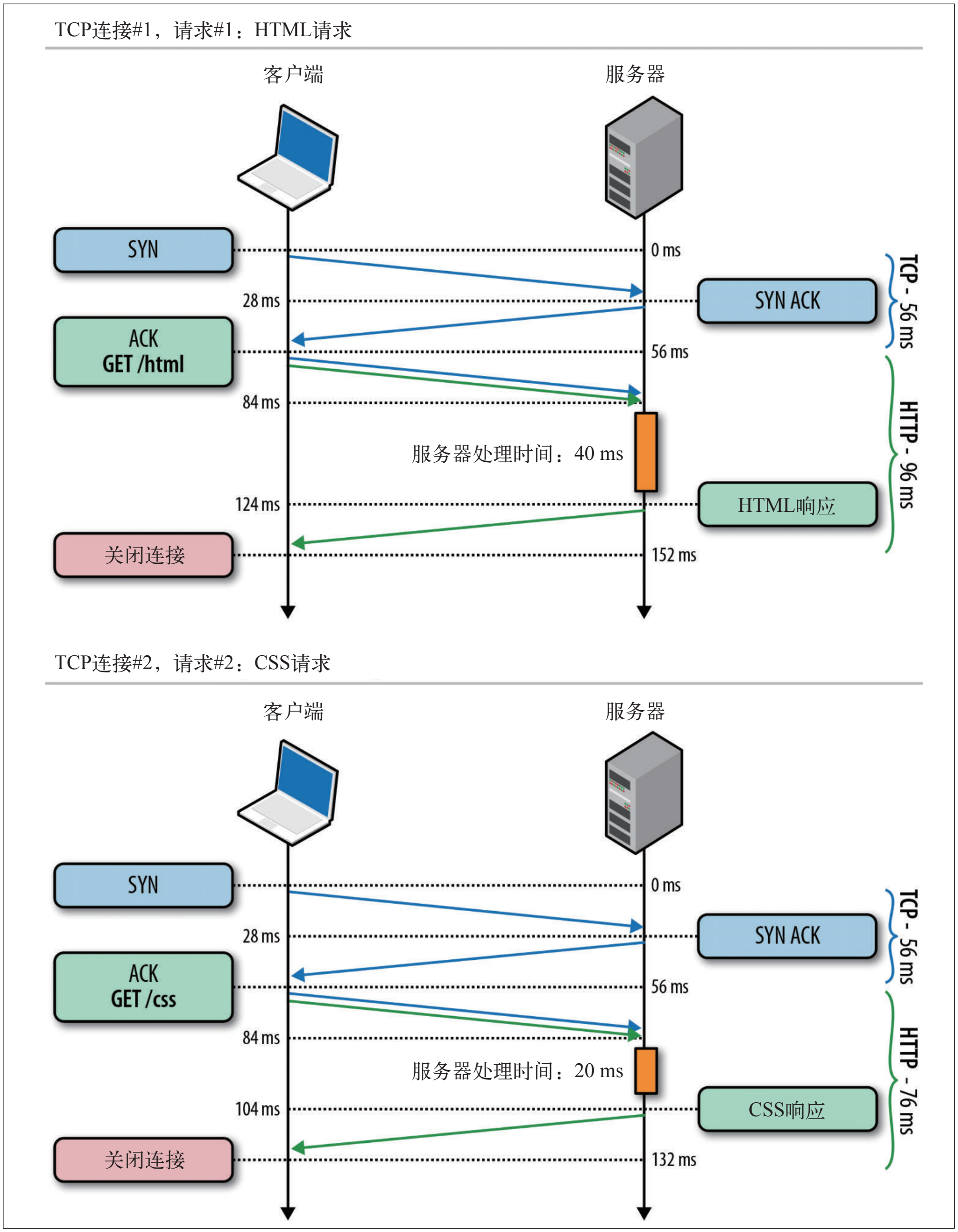


图 11-1：通过单独的 TCP 连接取得 HTML 和 CSS 文件

> 服务器处理速度越快，固定延迟对每个网络请求总时间的影响就越大！要验证这一点，可以改一改前面例子中的往返时间和服务器处理时间。

实际上，这时候最简单的优化就是重用底层的连接！添加对 HTTP 持久连接的支持，就可以避免第二次 TCP 连接时的三次握手、消除另一次 TCP 慢启动的往返，节约整整一次网络延迟。

在我们两个请求的例子中，总共只节约了一次往返时间。但是，更常见的情况是一次 TCP 连接要发送 *N* 次 HTTP 请求，这时：

- 没有持久连接，每次请求都会导致两次往返延迟；
- 有持久连接，只有第一次请求会导致两次往返延迟，后续请求只会导致一次往返延迟。

在启用持久连接的情况下，*N* 次请求节省的总延迟时间就是（*N*–1）×RTT。还记得吗，前面说过，在当代 Web 应用中，*N* 的平均值是 90，而且还在继续增加（10.2 节“剖析现代 Web 应用”）。因此，依靠持久连接节约的时间，很快就可以用秒来衡量了！这充分说明持久化 HTTP 是每个 Web 应用的关键优化手段。

客户端和服务器上的连接重用

好消息是，只要服务器愿意配合，所有现代浏览器都会尝试使用持久化 HTTP 连接。可以检查一下自己的应用和代理服务器配置，确保使用持久连接。为保证最好的结果，请使用 HTTP 1.1，因为它默认启用持久连接。如果只能使用 HTTP 1.0，则可以明确使用 `Connection: Keep-Alive` 首部声明使用持久连接。

此外，还要注意 HTTP 库和框架的默认行为，因为很多库和框架经常会默认使用非持久连接，这种做法多数源于它们提供“更简单 API”的理念。只要使用原始 HTTP 连接，一定记得重用它们：重用连接的性能提升非常巨大！

11.2 HTTP管道

持久 HTTP 可以让我们重用已有的连接来完成多次应用请求，但多次请求必须严格满足先进先出（FIFO）的队列顺序：发送请求，等待响应完成，再发送客户端队列中的下一个请求。HTTP 管道是一个很小但对上述工作流却非常重要的一次优化。管道可以让我们把 FIFO 队列从客户端（请求队列）迁移到服务器（响应队列）。

要理解这样做的好处，我们再看一看图 11-2。首先，服务器处理完第一次请求后，会发生了一次完整的往返：先是响应回传，接着是第二次请求。在此期间服务器空闲。如果服务器能在处理完第一次请求后，立即开始处理第二次请求呢？甚至，如果服务器可以并行或在多线程上或者使用多个工作进程，同时处理两个请求呢？

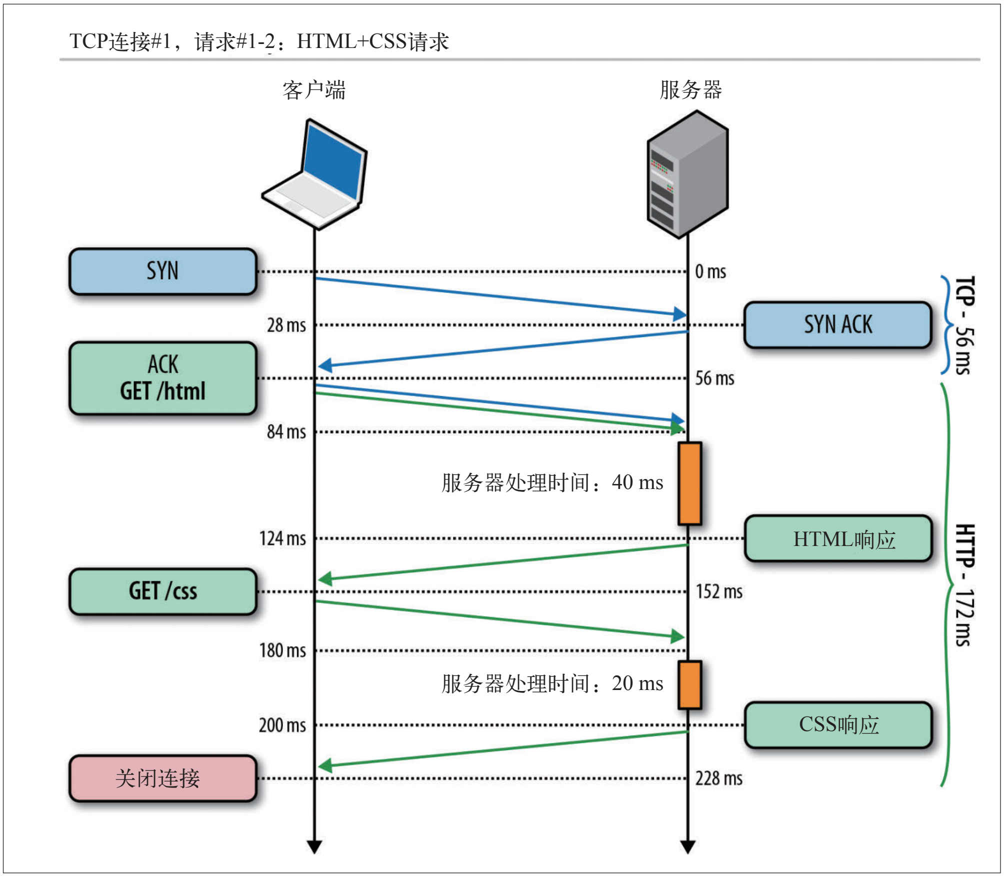


图 11-2：通过持久 TCP 连接取得 HTML 和 CSS 文件

通过尽早分派请求，不被每次响应阻塞，可以再次消除额外的网络往返。这样，就从非持久连接状态下的每个请求两次往返，变成了整个请求队列只需要两次网络往返（图 11-3）！

> HTTP 1.1 管道的好处，主要就是消除了发送请求和响应的等待时间。这种并行处理请求的能力对提升应用性能的帮助非常之大。

现在我们暂停一会，回顾一下在性能优化方面的收获。一开始，每个请求要用两个 TCP 连接（图 11-1），总延迟为 284 ms。在使用持久连接后（图 11-2），避免了一次握手往返，总延迟减少为 228 ms。最后，通过使用 HTTP 管道，又减少了两次请求之间的一次往返，总延迟减少为 172 ms。这样，从 284 ms 到 172 ms，这 40% 的性能提升完全拜简单的协议优化所赐。

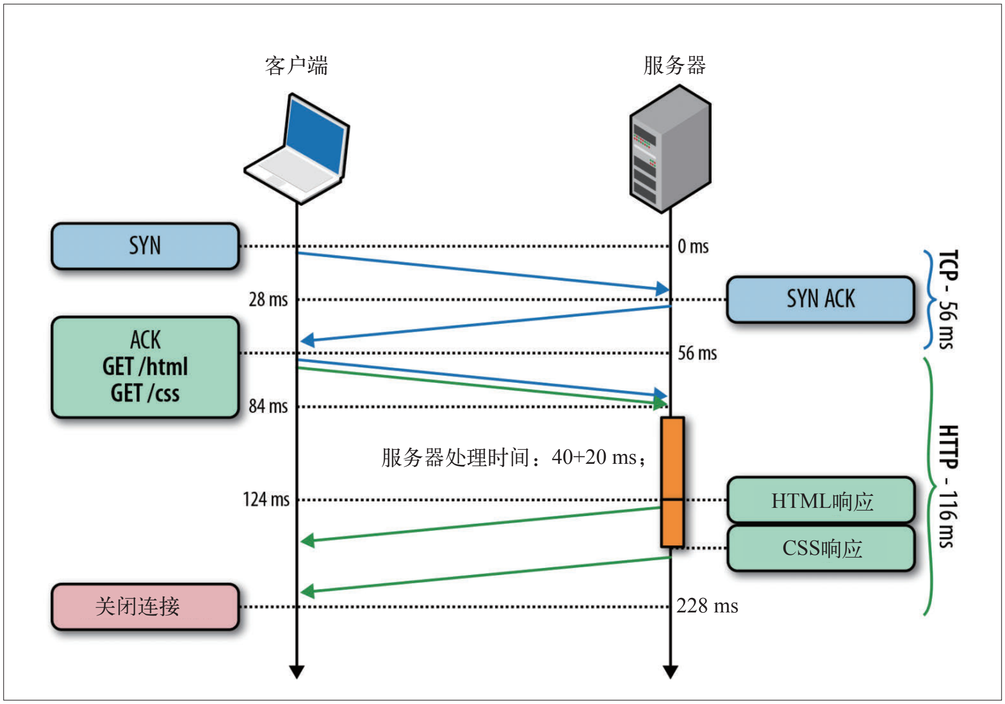


图 11-3：使用 HTTP 管道发送请求，服务器端按 FIFO 处理队列

而且，这 40% 的性能提升还不是固定不变的。这个数字与我们选择的网络延迟和两个请求的例子有关。希望读者自己能够尝试一些不同的情况，比如延迟更高、请求更多的情况。尝试之后，你会惊讶于性能提升效果比这里还要高得多。事实上，网络延迟越高，请求越多，节省的时间就越多。我觉得大家很有必要自己动手验证一下这个结果。因此，越是大型应用，网络优化的影响越大。

不过，这还不算完。眼光敏锐的读者可能已经发现了，我们可以在服务器上并行处理请求。理论上讲，没有障碍可以阻止服务器同时处理管道中的请求，从而再减少 20 ms 的延迟。

可惜的是，当我们想要采取这个优化措施时，发现了 HTTP 1.x 协议的一些局限性。HTTP 1.x 只能严格串行地返回响应。特别是，HTTP 1.x 不允许一个连接上的多个响应数据交错到达（多路复用），因而一个响应必须完全返回后，下一个响应才会开始传输。为说明这一点，我们可以看看服务器并行处理请求的情况（图 11-4）。

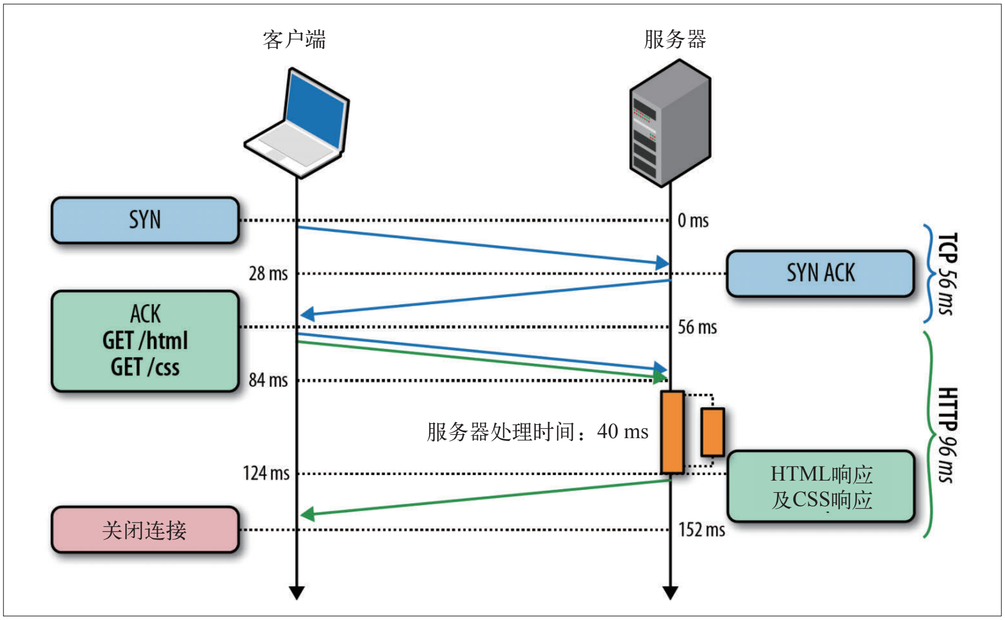


图 11-4：使用 HTTP 管道发送请求，且服务器端并行处理

图 11-4 演示了如下几个方面：

- HTML 和 CSS 请求同时到达，但先处理的是 HTML 请求；
- 服务器并行处理两个请求，其中处理 HTML 用时 40 ms，处理 CSS 用时 20 ms；
- CSS 请求先处理完成，但被缓冲起来以等候发送 HTML 响应；
- 发送完 HTML 响应后，再发送服务器缓冲中的 CSS 响应。

即使客户端同时发送了两个请求，而且 CSS 资源先准备就绪，服务器也会先发送 HTML 响应，然后再交付 CSS。这种情况通常被称作*队首阻塞*，并经常导致次优化交付：不能充分利用网络连接，造成服务器缓冲开销，最终导致无法预测的客户端延迟。假如第一个请求无限期挂起，或者要花很长时间才能处理完，怎么办呢？在 HTTP 1.1 中，所有后续的请求都将被阻塞，等待它完成。

> 在前面讨论 TCP 时，我们已经提到过队首阻塞的问题了。由于 TCP 要求严格按照顺序交付，丢失一个 TCP 分组就会阻塞所有高序号的分组，除非重传那个丢失的分组，这样就会导致额外的应用延迟，详细情况请参考 2.4 节“队首阻塞”。

实际中，由于不可能实现多路复用，HTTP 管道会导致 HTTP 服务器、代理和客户端出现很多微妙的，不见文档记载的问题：

- 一个慢响应就会阻塞所有后续请求；
- 并行处理请求时，服务器必须缓冲管道中的响应，从而占用服务器资源，如果有个响应非常大，则很容易形成服务器的受攻击面；
- 响应失败可能终止 TCP 连接，从页强迫客户端重新发送对所有后续资源的请求，导致重复处理；
- 由于可能存在中间代理，因此检测管道兼容性，确保可靠性很重要；
- 如果中间代理不支持管道，那它可能会中断连接，也可能会把所有请求串联起来。

由于存在这些以及其他类似的问题，而 HTTP 1.1 标准中也未对此做出说明，HTTP 管道技术的应用非常有限，虽然其优点毋庸置疑。今天，一些支持管道的浏览器，通常都将其作为一个高级配置选项，但大多数浏览器都会禁用它。换句话说，如果浏览器是 Web 应用的主要交付工具，那还是很难指望通过 HTTP 管道来提升性能。

在浏览器外部使用 HTTP 管道

在完全忽略 HTTP 管道的优点之前，有必要提醒一下大家，如果你对客户端和服务器拥有完全控制的权限，那么还是可以使用它的，并且效果非常好。事实上，苹果 iTunes 那个案例就说明了问题，参见本章开头的“让 iTunes 用户感受到 3 倍以上的性能增强”。如此巨大的性能提升，就来自启用持久 HTTP 连接，以及在服务器和 iTunes 客户端内启用 HTTP 管道。

要在你自己的应用中启用管道，要注意如下事项：

- 确保 HTTP 客户端支持管道；
- 确保 HTTP 服务器支持管道；
- 应用必须处理中断的连接并恢复；
- 应用必须处理中断请求的幂等问题；
- 应用必须保护自身不受出问题的代理的影响。

实践中部署 HTTP 管道的最佳途径，就是在客户端和服务器间使用安全通道（HTTPS）。这样，就能可靠地避免那些不理解或不支持管道的中间代理的干扰。

11.3　使用多个TCP连接

由于 HTTP 1.x 不支持多路复用，浏览器可以不假思索地在客户端排队所有 HTTP 请求，然后通过一个持久连接，一个接一个地发送这些请求。然而，这种方式在实践中太慢。实际上，浏览器开发商没有别的办法，只能允许我们并行打开多个 TCP

会话。多少个？现实中，大多数现代浏览器，包括桌面和移动浏览器，都支持每个主机打开 6 个连接。

进一步讨论之前，有必要先想一想同时打开多个 TCP 连接意味着什么。当然，有正面的也有负面的。下面我们以每个主机打开最多 6 个独立连接为例：

- 客户端可以并行分派最多 6 个请求；
- 服务器可以并行处理最多 6 个请求；
- 第一次往返可以发送的累计分组数量（TCP cwnd）增长为原来的 6 倍。

在没有管道的情况下，最大的请求数与打开的连接数相同。相应地，TCP 拥塞窗口也要乘以打开的连接数量，从而允许客户端绕开由 TCP 慢启动规定的分组限制。这好像是一个方便的解决方案。我们再看看这样做的代价：

- 更多的套接字会占用客户端、服务器以及代理的资源，包括内存缓冲区和 CPU 时钟周期；
- 并行 TCP 流之间竞争共享的带宽；
- 由于处理多个套接字，实现复杂性更高；
- 即使并行 TCP 流，应用的并行能力也受限制。

实践中，CPU 和内存占用并非微不足道，由此会导致客户端和服务器端的资源占用量上升，运维成本提高。类似地，由于客户端实现的复杂性提高，开发成本也会提高。最后，说到应用的并行性，这种方式提供的好处还是非常有限的。这不是一个长期的方案。了解这些之后，可以说今天之所以使用它，主要有三个原因：

(1) 作为绕过应用协议（HTTP）限制的一个权宜之计；
(2) 作为绕过 TCP 中低起始拥塞窗口的一个权宜之计；
(3) 作为让客户端绕过不能使用 TCP 窗口缩放的一个权宜之计（参见 2.3 节“带宽延迟积”）。

后两个针对 TCP 的问题（窗口缩放和 cwnd）最好是通过升级到最新的 OS 内核来解决，参见 2.5 节“针对 TCP 的优化建议”。cwnd 值最近又提高到了 10 个分组，而所有最新的平台都能可靠地支持 TCP 窗口缩放。这当然是好消息。但坏消息是，没有更好办法绕开 HTTP 1.x 的多路复用问题。

只要必须支持 HTTP 1.x 客户端，就不得不想办法应对多 TCP 流的问题。而这又会带来一个明显的问题：为什么浏览器要规定每个主机 6 个连接呢？恐怕有读者也猜到了，这个数字是多方平衡的结果：这个数字越大，客户端和服务器的资源占用越多，但随之也会带来更高的请求并行能力。每个主机 6 个连接只不过是大家都觉得

比较安全的一个数字。对某些站点而言，这个数字已经足够了，但对其他站点来说，可能还满足不了需求。

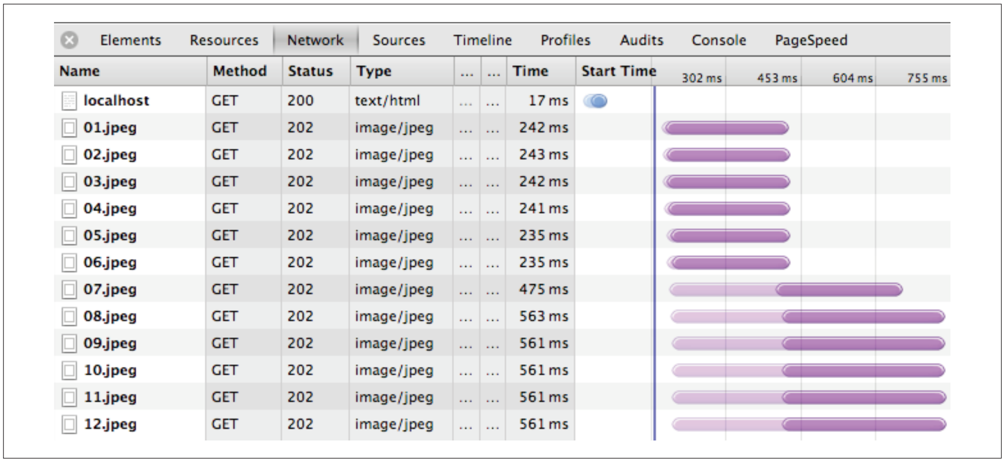
> **消耗客户端和服务器资源**
>
> 限制每个主机最多 6 个连接，可以让浏览器检测出无意（或有意）的 DoS（Denial of Service）攻击。如果没有这个限制，客户端有可能消耗掉服务器的所有资源。
>
> 讽刺的是，同样的安全检测在某些浏览器上却会招致反向攻击：如果客户端超过了最大连接数，那么所有后来的客户端请求都将被阻塞。大家可以做个试验，在一个主机上同时打开 6 个并行下载，然后再打开第 7 个下载请求，这个请求会挂起，直到前面的请求完成才会执行。
>
> 用足客户端连接的限制似乎是一个可以接受的安全问题，但对于需要实时交付数据的应用而言，这样做越来越容易造成部署上的问题。比如 WebSocket、Server Sent Event 和挂起 XHR，这些会话都会占用整整一个 TCP 流，而不管有无数据传输——记住，没有多路复用一说！实际上，如果你不注意，那很可能自己对自己的应用施加 DoS 攻击。

11.4 域名分区

HTTP 1.x 协议的一项空白强迫浏览器开发商引入并维护着连接池，每个主机最多 6 个 TCP 流。好的一方面是对这些连接的管理工作都由浏览器来处理。作为应用开发者，你根本不必修改自己的应用。不好的一方面呢，就是 6 个并行的连接对你的应用来说可能仍然不够用。

根据 HTTP Archive 的统计，目前平均每个页面都包含 90 多个独立的资源，如果这些资源都来自同一个主机，那么仍然会导致明显的排队等待（图 11-5）。实际上，何必把自己只限制在一个主机上呢？我们不必只通过一个主机（例如 www.example.com）提供所有资源，而是可以手工将所有资源分散到多个子域名：{shard1, shardn}.example.com。由于主机名称不一样了，就可以突破浏览器的连接限制，实现更高的并行能力。域名分区使用得越多，并行能力就越强！

当然，天下没有免费的午餐，域名分区也不例外：每个新主机名都要求有一次额外的 DNS 查询，每多一个套接字都会多消耗两端的一些资源，而更糟糕的是，站点作者必须手工分离这些资源，并分别把它们托管到多个主机上。



Name	Method	Status	Type	...	...	Time
localhost	GET	200	text/html	...	...	17 ms
01.jpeg	GET	202	image/jpeg	...	...	242 ms
02.jpeg	GET	202	image/jpeg	...	...	243 ms
03.jpeg	GET	202	image/jpeg	...	...	242 ms
04.jpeg	GET	202	image/jpeg	...	...	241 ms
05.jpeg	GET	202	image/jpeg	...	...	235 ms
06.jpeg	GET	202	image/jpeg	...	...	235 ms
07.jpeg	GET	202	image/jpeg	...	...	475 ms
08.jpeg	GET	202	image/jpeg	...	...	563 ms
09.jpeg	GET	202	image/jpeg	...	...	561 ms
10.jpeg	GET	202	image/jpeg	...	...	561 ms
11.jpeg	GET	202	image/jpeg	...	...	561 ms
12.jpeg	GET	202	image/jpeg	...	...	561 ms

图 11-5：由于每个主机只能同时发起 6 个连接而导致的资源错列

实践中，把多个域名（如 shard1.example.com、shard2.example.com）解析到同一个 IP 地址是很常见的做法。所有分区都通过 CNAME DNS 记录指向同一个服务器，而浏览器连接限制针对的是主机名，不是 IP 地址。另外，每个分区也可以指向一个 CDN 或其他可以访问到的服务器。

怎么计算最优的分区数目呢？这个问题不好回答，因为没有简单的方程式。答案取决于页面中资源的数量（每个页面都可能不一样），以及客户端连接的可用带宽和延迟（因客户端而异）。实际上，我们能做的，就是在调查的基础上做出预测，然后使用固定数量的分区。幸运的话，多这么一点复杂性，还是能给大多数用户带来好处的。

实践中，域名分区经常会被滥用，导致几十个 TCP 流都得不到充分利用，其中很多永远也避免不了 TCP 慢启动，最坏的情况下还会降低性能。此外，如果使用的是 HTTPS，那么由于 TLS 握手导致的额外网络往返，会使得上述代价更高。此时，请大家注意如下几条：

- 首先，把 TCP 利用好，参见 2.5 节“针对 TCP 的优化建议”；
- 浏览器会自动为你打开 6 个连接；
- 资源的数量、大小和响应时间都会影响最优的分区数目；
- 客户端延迟和带宽会影响最优的分区数目；
- 域名分区会因为额外的 DNS 查询和 TCP 慢启动而影响性能。

域名分区是一种合理但又不完美的优化手段。请大家一定先从最小分区数目（不分区）开始，然后逐个增加分区并度量分区后对应用的影响。现实当中，真正因同时打开十几个连接而提升性能的站点并不多，如果你最终使用了很多分区，那么你会

发现减少资源数量或者将它们合并为更少的请求，反而能带来更大的好处。

> DNS 查询和 TCP 慢启动导致的额外消耗对高延迟客户端的影响最大。换句话说，移动（3G、4G）客户端经常是受过度域名分区影响最大的！

11.5 度量和控制协议开销

HTTP 0.9 当初就是一个简单的只有一行的 ASCII 请求，用于取得一个超文本文档，这样导致的开销是最小的。HTTP 1.0 增加了请求和响应首部，以便双方能够交换有关请求和响应的元信息。最终，HTTP 1.1 把这种格式变成了标准：服务器和客户端都可以轻松扩展首部，而且始终以纯文本形式发送，以保证与之前 HTTP 版本的兼容。

今天，每个浏览器发起的 HTTP 请求，都会携带额外 500~800 字节的 HTTP 元数据：用户代理字符串、很少改变的接收和传输首部、缓存指令，等等。有时候，500~800 字节都少说了，因为没有包含最大的一块：HTTP cookie。现代应用经常通过 cookie 进行会话管理、记录个性选项或者完成分析。综合到一起，所有这些未经压缩的 HTTP 元数据经常会给每个 HTTP 请求增加几千字节的协议开销。

> RFC 2616（HTTP 1.1）没有对 HTTP 首部的大小规定任何限制。然而，实际中，很多服务器和代理都会将其限制在 8 KB 或 16 KB 之内。

HTTP 首部的增多对它本身不是坏事，因为大多数首部都有其特定用途。然而，由于所有 HTTP 首部都以纯文本形式发送（不会经过任何压缩），这就会给每个请求附加较高的额外负荷，而这在某些应用中可能造成严重的性能问题。举个例子，API 驱动的 Web 应用越来越多，这些应用需要频繁地以序列化消息（如 JSON）的形式通信。在这些应用中，额外的 HTTP 开销经常会超过实际传输的数据静荷一个数量级：

```
$> curl --trace-ascii - -d'{"msg":"hello"}' http://www.igvita.com/api

== Info: Connected to www.igvita.com
=> Send header, 218 bytes ❶
POST /api HTTP/1.1
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 ...
Host: www.igvita.com
Accept: */*
Content-Length: 15 ❷
```

```
Content-Type: application/x-www-form-urlencoded
=> Send data, 15 bytes (0xf)
{"msg":"hello"}

<= Recv header, 134 bytes ❸
HTTP/1.1 204 No Content
Server: nginx/1.0.11
Via: HTTP/1.1 GWA
Date: Thu, 20 Sep 2012 05:41:30 GMT
Cache-Control: max-age=0, no-cache
```

❶ HTTP 请求首部：218 字节

❷ 应用静荷 15 字节（`{"msg":"hello"}`）

❸ 服务器的 204 响应：134 字节

在前面的例子中，寥寥 15 个字符的 JSON 消息被 352 字节的 HTTP 首部包裹着，全部以纯文本形式发送——协议字节开销占 96%，而且这还是没有 cookie 的最好情况。减少要传输的首部数据（高度重复且未压缩），可以节省相当于一次往返的延迟时间，显著提升很多 Web 应用的性能。

> Cookie 在很多应用中都是常见的性能瓶颈，很多开发者都会忽略它给每次请求增加的额外负担。相关讨论请参见 13.1.3 节“消除不必要的请求字节”。

11.6 连接与拼合

最快的请求是不用请求。不管使用什么协议，也不管是什么类型的应用，减少请求次数总是最好的性能优化手段。可是，如果你无论如何也无法减少请求，那么对 HTTP 1.x 而言，可以考虑把多个资源捆绑打包到一块，通过一次网络请求获取：

- 连接

  把多个 JavaScript 或 CSS 文件组合为一个文件。

- 拼合

  把多张图片组合为一个更大的复合的图片。

对 JavaScript 和 CSS 来说，只要保持一定的顺序，就可以做到把多个文件连接起来而不影响代码的行为和执行。类似地，多张图片可以组合为一个“图片精灵”，然后使用 CSS 选择这张大图中的适当部分，显示在浏览器中。这两种技术都具备两方面的优点。

- *减少协议开销*

  通过把文件组合成一个资源，可以消除与文件相关的协议开销。如前所述，每个文件很容易招致 KB 级未压缩数据的开销。

- *应用层管道*

  说到传输的字节，这两种技术的效果都好像是启用了 HTTP 管道：来自多个响应的数据前后相继地连接在一起，消除了额外的网络延迟。实际上，就是把管道提高了一层，置入了应用中。

连接和拼合技术都属于以内容为中心的应用层优化，它们通过减少网络往返开销，可以获得明显的性能提升。可是，实现这些技术也要求额外的处理、部署和编码（比如选择图片精灵中子图的 CSS 代码），因而也会给应用带来额外的复杂性。此外，把多个资源打包到一块，也可能给缓存带来负担，影响页面的执行速度。

要理解为什么这些技术会伤害性能，可以考虑一种并不少见的情况：一个包含十来个 JavaScript 和 CSS 文件的应用，在产品状态下把所有文件合并为一个 CSS 文件和一个 JavaScript 文件。

- 相同类型的资源都位于一个 URL（缓存键）下面。
- 资源包中可能包含当前页面不需要的内容。
- 对资源包中任何文件的更新，都要求重新下载整个资源包，导致较高的字节开销。
- JavaScript 和 CSS 只有在传输完成后才能被解析和执行，因而会拖慢应用的执行速度。

实践中，大多数 Web 应用都不是只有一个页面，而是由多个视图构成。每个视图都有自己的资源，同时资源之间还有部分重叠：公用的 CSS、JavaScript 和图片。实际上，把所有资源都组合到一个文件经常会导致处理和加载不必要的字节。虽然可以把它看成一种预获取，但代价则是降低了初始启动的速度。

对很多应用来说，更新资源带来的问题更大。更新图片精灵或组合 JavaScript 文件中的某一处，可能就会导致重新传输几百 KB 数据。由于牺牲了模块化和缓存粒度，假如打包资源变动频率过高，特别是在资源包过大的情况下，很快就会得不偿失。如果你的应用真到了这种境地，那么可以考虑把“稳定的核心”，比如框架和库，转移到独立的包中。

内存占用也会成为问题。对图片精灵来说，浏览器必须分析整个图片，即便实际上只显示了其中的一小块，也要始终把整个图片都保存在内存中。浏览器是不会把不显示的部分从内存中剔除掉的！

计算图片对内存的需求

所有编码的图片经浏览器解析后都会以 RGBA 位图的形式保存于内存当中。每个 RGBA 图片的像素需要占用 4 字节：红、绿、蓝通道各占 1 字节，Alpha（透明）通道占 1 字节。这样算下来，一张图片占用的内存量就是图片像素宽度 × 像素高度 ×4 字节。

举个例子，800×600 像素的位图会占多大内存呢？

800 × 600 × 4 B = 1 920 000 B ≈ 1.83 MB

在资源受限的设备，比如手机上，内存占用很快就会成为瓶颈。对于游戏等严重依赖图片的应用来说，这个问题就会更明显。

最后，为什么执行速度还会受影响呢？我们知道，浏览器是以递增方式处理 HTML 的，而对于 JavaScript 和 CSS 的解析及执行，则要等到整个文件下载完毕。JavaScript 和 CSS 处理器都不允许递增式执行。

CSS 和 JavaScript 文件大小与执行性能

CSS 文件越大，浏览器在构建 CSSOM 前经历的阻塞时间就越长，从而推迟首次绘制页面的时间。类似地，JavaScript 文件越大，对执行速度的影响同样越大；小文件倒是能实现“递增式”执行。

打包文件到底多大合适呢？可惜的是，没有理想的大小。然而，谷歌 PageSpeed 团队的测试表明，30~50 KB（压缩后）是每个 JavaScript 文件大小的合适范围：既大到了能够减少小文件带来的网络延迟，还能确保递增及分层式的执行。具体的结果可能会由于应用类型和脚本数量而有所不同。

总之，连接和拼合是在 HTTP 1.x 协议限制（管道没有得到普遍支持，多请求开销大）的现实之下可行的应用层优化。使用得当的话，这两种技术可以带来明显的性能提升，代价则是增加应用的复杂度，以及导致缓存、更新、执行速度，甚至渲染页面的问题。应用这两种优化时，要注意度量结果，根据实际情况考虑如下问题。

- 你的应用在下载很多小型的资源时是否会被阻塞？
- 有选择地组合一些请求对你的应用有没有好处？
- 放弃缓存粒度对用户有没有负面影响？
- 组合图片是否会占用过多内存？
- 首次渲染时是否会遭遇延迟执行？

在上述问题的答案间求得平衡是一种艺术。

优化 Gmail 性能

Gmail 使用了大量 JavaScript，而且也不断拓展了现代浏览器的性能边界。要提升首次加载性能，Gmail 团队尝试了各种技术，目前包括如下这些：

- 把首次绘制所需的 CSS 单独拿出来，优先于其他 CSS 文件发送；
- 递增地交付较小的 JavaScript 块，以实现递增式执行；
- 使用定制的外部更新机制，即客户端在后台下载新的 JavaScript 文件，然后在页面刷新时更新。

鉴于 Gmail 如此庞大的用户数量，如果所有打开的浏览器都要更新脚本，那哪怕一次简单的 JavaScript 更新，都可能演变为一次自残式的 DoS 攻击。为此，Gmail 会在用户使用旧版本页面时，在后台预先加载更新文件，这样既可以分散负荷，又能提升下一次刷新时的速度。这个过程每天都重复不止一次。

在此基础上，为了让用户感觉第一次加载的速度很快，Gmail 团队还在 HTML 文档中嵌入了关键性 CSS 和 JavaScript，然后以块的形式递增加载其余 JavaScript 文件，以加快脚本执行——第一次打开 Gmail 时显示的进度条，反映的就是这个过程！

11.7　嵌入资源

嵌入资源是另一种非常流行的优化方法，把资源嵌入文档可以减少请求的次数。比如，JavaScript 和 CSS 代码，通过适当的 `script` 和 `style` 块可以直接放在页面中，而图片甚至音频或 PDF 文件，都可以通过数据 URI（`data:[mediatype][;base64],data`）的方式嵌入到页面中：

```
<img src="data:image/gif;base64,R0lGODlhAQABAIAAAAA
        AAAAAACH5BAAAAAAALAAAAAABAAEAAAICTAEAOw=="
     alt="1x1 transparent (GIF) pixel" />
```

> 前面的例子是在文档中嵌入了一个 1×1 的透明 GIF 像素。而任何 MIME 类型，只要浏览器能理解，都可以通过类似方式嵌入到页面中，包括 PDF、音频、视频。不过，有些浏览器会限制数据 URI 的大小，比如 IE8 最大只允许 32 KB。

数据 URI 适合特别小的，理想情况下，最好是只用一次的资源。以嵌入方式放到页面中的资源，应该算是页面的一部分，不能被浏览器、CDN 或其他缓存代理作为单独的资源缓存。换句话说，如果在多个页面中都嵌入同样的资源，那么这个资源将会随着每个页面的加载而被加载，从而增大每个页面的总体大小。另外，如果嵌入

资源被更新，那么所有以前出现过它的页面都将被宣告无效，而由客户端重新从服务器获取。

最后，虽然 CSS 和 JavaScript 等基于文本的资源很容易直接嵌入页面，也不会带来多余的开销，但非文本性资源则必须通过 base64 编码，而这会导致开销明显增大：编码后的资源大小比原大小增大 33%！

> base64 编码使用 64 个 ASCII 符号和空白符将任意字节流编码为 ASCII 字符串。编码过程中，base64 会导致被编码的流变成原来的 4/3，即增大 33% 的字节开销。

实践中，常见的一个经验规则是只考虑嵌入 1~2 KB 以下的资源，因为小于这个标准的资源经常会导致比它自身更高的 HTTP 开销。然而，如果嵌入的资源频繁变更，又会导致宿主文档的无效缓存率升高。嵌入资源也不是完美的方法。如果你的应用要使用很小的、个别的文件，在考虑是否嵌入时，可以参照如下建议：

- 如果文件很小，而且只有个别页面使用，可以考虑嵌入；
- 如果文件很小，但需要在多个页面中重用，应该考虑集中打包；
- 如果小文件经常需要更新，就不要嵌入了；
- 通过减少 HTTP cookie 的大小将协议开销最小化。

第12章 HTTP 2.0

HTTP 2.0 可以让我们的应用更快、更简单、更健壮——这几词凑到一块是很罕见的！ HTTP 2.0 把很多以前我们针对 HTTP 1.1 想出来的“歪招儿”一笔勾销，把解决那些问题的方案内置在了传输层中。不仅如此，HTTP 2.0 还为我们进一步优化应用、改进性能，提供了全新的机会！

HTTP 2.0 的目的就是通过支持请求与响应的多路复用来减少延迟，通过压缩 HTTP 首部字段将协议开销降至最低，同时增加对请求优先级和服务器端推送的支持。为达成这些目标，HTTP 2.0 还会给我们带来大量其他协议层面的辅助实现，比如新的流量控制、错误处理和更新机制。上述几种机制虽然不是全部，但却是最重要的，所有 Web 开发者都应该理解并在自己的应用中利用它们。

HTTP 2.0 不会改动 HTTP 的语义。HTTP 方法、状态码、URI 及首部字段，等等这些核心概念一如往常。但是，HTTP 2.0 修改了格式化数据（分帧）的方式，以及客户端与服务器间传输这些数据的方式。这两点统帅全局，通过新的组帧机制向我们的应用隐藏了所有复杂性。换句话说，所有原来的应用都可以不必修改而在新协议运行。这当然是好事。

可是，我们关心的不止是交付能用的应用，我们目标是交付最佳性能！ HTTP 2.0 为我们的应用提供了很多新的优化机制，这些机制是前所未有的，而我们的工作就是把它们都利用好。下面我们就来详细介绍一下这些机制。

开发中的标准

HTTP 2.0 还在积极的开发过程中，其核心架构设计、原理及特性非常完善，但这不代表具体的、底层的实现也同样如此。所以，我们的讨论将围绕架构及其意义展开，同时简要介绍一下数据格式。这些对理解新协议的原理和用途已经够了。

要了解 HTTP 2.0 标准的最新草案和状态，请访问 IETF 的跟踪页面：http://tools.ietf.org/html/draft-ietf-httpbis-http2。

12.1 历史及其与SPDY的渊源

SPDY 是谷歌开发的一个实验性协议，于 2009 年年中发布，其主要目标是通过解决 HTTP 1.1 中广为人知的一些性能限制，来减少网页的加载延迟。大致上，这个项目设定的目标如下：

- 页面加载时间（PLT，Page Load Time）降低 50%；
- 无需网站作者修改任何内容；
- 把部署复杂性降至最低，无需变更网络基础设施；
- 与开源社区合作开发这个新协议；
- 收集真实性能数据，验证这个实验性协议是否有效。

为了达到降低 50% 页面加载时间的目标，SPDY 引入了一个新的二进制分帧数据层，以实现多向请求和响应、优先次序、最小化及消除不必要的网络延迟，目的是更有效地利用底层 TCP 连接；参见 10.3.2 节“延迟是性能瓶颈”。

首次发布后不久，谷歌的两位软件工程师 Mike Belshe 和 Roberto Peon 就分享了他们对这个新实验性 SPDY 协议的实现结果、文档和源代码：

> 目前为止，我们只在实验室条件下测试过 SPDY。最初的成果很激动人心：通过模拟的家庭上网线路下载了 25 个最流行的网站之后，我们发现性能的改进特别明显，页面加载速度最多快了 55%。
>
> ——A 2x Faster Web Chromium Blog

几年后的 2012 年，这个新的实验性协议得到了 Chrome、Firefox 和 Opera 的支持，很多大型网站（如谷歌、Twitter、Facebook）都对兼容客户端提供 SPDY 会话。换句话说，SPDY 在被行业采用并证明能够大幅提升性能之后，已经具备了成为一个标准的条件。最终，HTTP-WG（HTTP Working Group）在 2012 年初把 HTTP 2.0 提到了议事日程，吸取 SPDY 的经验教训，并在此基础上制定官方标准。

12.2　走向HTTP 2.0

SPDY 是 HTTP 2.0 的催化剂，但 SPDY 并非 HTTP 2.0。2012 年初，W3C 向社会征集 HTTP 2.0 的建议，HTTP-WG 经过内部讨论，决定将 SPDY 规范作为制定标准的基础。从那时起，SPDY 已经经过了很多变化和改进，而且在 HTTP 2.0 官方标准公布之前，还将有很多变化和改进。

在此，有必要回顾一下 HTTP 2.0 宣言草稿，因为这份宣言明确了该协议的范围和关键设计要求：

> HTTP/2.0 应该满足如下条件：
>
> - 相对于使用 TCP 的 HTTP 1.1，用户在大多数情况下的感知延迟要有实质上、可度量的改进；
> - 解决 HTTP 中的“队首阻塞”问题；
> - 并行操作无需与服务器建立多个连接，从而改进 TCP 的利用率，特别是拥塞控制方面；
> - 保持 HTTP 1.1 的语义，利用现有文档，包括（但不限于）HTTP 方法、状态码、URI，以及首部字段；
> - 明确规定 HTTP 2.0 如何与 HTTP 1.x 互操作，特别是在中间介质上；
> - 明确指出所有新的可扩展机制以及适当的扩展策略。
>
> 对现有的 HTTP 部署——特别是 Web 浏览器（桌面及移动）、非浏览器（“HTTP API”）、Web 服务（各种规模），以及中间介质（代理、公司防火墙、“反向”代理及 CDN）而言，最终规范应该满足上述这些目标。类似地，当前和未来对 HTTP/1.x 的语义扩展（如首部、方法、状态码、缓存指令）也应该得到新协议的支持。
>
> ——HTTPbis WG 宣言 HTTP 2.0

简言之，HTTP 2.0 致力于突破上一代标准众所周知的性能限制，但它也是对之前 1.x 标准的扩展，而非替代。HTTP 的语义不变，提供的功能不变，HTTP 方法、状态码、URI 和首部字段，等等这些核心概念也不变；这些方面的变化都不在考虑之列。既然如此，那“2.0”还名副其实吗？

之所以要递增一个大版本到 2.0，主要是因为它改变了客户端与服务器之间交换数据的方式。为实现宏伟的性能改进目标，HTTP 2.0 增加了新的二进制分帧数据层，而这一层并不兼容之前的 HTTP 1.x 服务器及客户端——是谓 2.0。

> 除非你在实现 Web 服务器或者定制客户端，需要使用原始的 TCP 套接字，否则你很可能注意不到 HTTP 2.0 技术面的实际变化：所有新的、低级分帧机制都是浏览器和服务器为你处理的。或许唯一的区别就是可选的 API 多了一些，比如服务器推送！

最后，有必要了解一下 HTTP 2.0 的进度。开发一个支撑所有 Web 通信协议的升级版可不是件小事，需要周密考虑、反复试验、多方协调。因此，猜测 HTTP 2.0 什么时候制定完成很不靠谱，我们只能说：到完成的时候自然就完成了。话虽这么说，但 HTTP-WG 的进度并不慢，下面是官方目前设置的里程碑。

- 2012 年 3 月：征集 HTTP 2.0 建议。
- 2012 年 9 月：HTTP 2.0 的第一个草案发布。
- 2013 年 7 月：HTTP 2.0 草案的第一个实现发布。
- 2014 年 4 月：工作组最后征集关于 HTTP 2.0 的意见。
- 2014 年 11 月：将 HTTP 2.0 作为建议标准提交给 IESG。

2012 年到 2014 年之间的主要工作集中在草案编辑和试验方面。根据进展情况，以及实现者和业界的反馈，整个进度可能会相应调整。好消息是，截止 2013 年初，这个计划进展顺利！

HTTP 2.0 与 SPDY 共同进化

2012 年夏天，HTTP 工作组采用了 SPDY v2 草案作为制定 HTTP 2.0 标准的起点。但 SPDY 的制定工作并没有因此停滞，相反它也还在并行进化：

- 2012 年发布的 SPDY v3 涵盖更新的帧格式和对流量控制的初次实现；
- 2013~2014 年即将发布的 SPDY v4 会再次更新帧格式，包含改进的优先级排定机制、流量控制及服务器推送的实现。

之所以还要继续制定 SPDY，原因很简单：把它作为 HTTP 2.0 新功能及新建议的实验场。纸上谈兵未必实际，而实际上可行的办法，未必能在撰写规范时想出来。SPDY 为 HTTP 2.0 标准收纳每一项建议，提供了事前的测试和评估手段。

SPDY 和 HTTP 2.0 的渐进发展和共同进化，给实现者带来了很大的工作量，但也带来很多好处：规范和客户端及服务器的实现，也能够并行进化，并得到可靠和广泛的测试。事实上，等到 HTTP 2.0 的状态变成“就绪”时，主流客户端和服务器都将内置经过了完备测试的实现！到那时，SPDY 就可以退役了，HTTP 2.0 将粉墨登场。

12.3　设计和技术目标

HTTP 1.x 的设计初衷主要是实现要简单：HTTP 0.9 只用一行协议就启动了万维网；HTTP 1.0 则是对流行的 0.9 扩展的一个正式说明；HTTP 1.1 则是 IETF 的一份官方标准（参见第 9 章）。因此，HTTP 0.9~1.x 只描述了现实是怎么一回事：HTTP 是应用最广泛、采用最多的一个互联网应用协议。

然而，实现简单是以牺牲应用性能为代价的，而这正是 HTTP 2.0 要致力于解决的：

> HTTP/2.0 通过支持首部字段压缩和在同一连接上发送多个并发消息，让应用更有效地利用网络资源，减少感知的延迟时间。而且，它还支持服务器到客户端的主动推送机制。
>
> ——HTTP/2.0，Draft 4

HTTP 2.0 当前还在制定过程中，因此如何编码每一帧中的比特、每个字段叫什么名字，以及类似的底层细节都可能会变化。但是，尽管“怎么”会不断演进，可核心设计和技术目标却是可以讨论的。这些内容已经取得了多方的理解和共识。

12.3.1　二进制分帧层

HTTP 2.0 性能增强的核心，全在于新增的二进制分帧层（图 12-1），它定义了如何封装 HTTP 消息并在客户端与服务器之间传输。

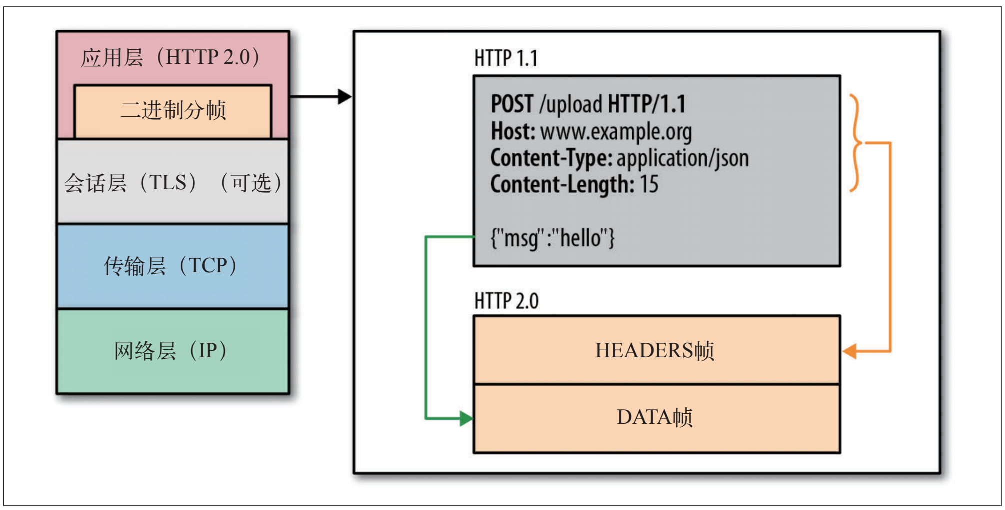


图 12-1：HTTP 2.0 二进制分帧层

这里所谓的“层”，指的是位于套接字接口与应用可见的高层 HTTP API 之间的一个新机制：HTTP 的语义，包括各种动词、方法、首部，都不受影响，不同的是传输期间对它们的编码方式变了。HTTP 1.x 以换行符作为纯文本的分隔符，而 HTTP 2.0 将所有传输的信息分割为更小的消息和帧，并对它们采用二进制格式的编码。

这样一来，客户端和服务器为了相互理解，必须都使用新的二进制编码机制：HTTP 1.x 客户端无法理解只支持 HTTP 2.0 的服务器，反之亦然。不过不要紧，现有的应用不必担心这些变化，因为客户端和服务器会替它们完成必要的分帧工作。

HTTPS 是二进制分帧的另一个典型示例：所有 HTTP 消息都以透明的方式为我们编码和解码（参见 4.6 节“TLS 记录协议”），从而实现客户端与服务器安全通信，但不必对应用进行任何修改。HTTP 2.0 的工作原理差不多也是这样。

12.3.2　流、消息和帧

新的二进制分帧机制改变了客户端与服务器之间交互数据的方式（图 12-2）。为了说明这个过程，我们需要了解 HTTP 2.0 的两个新概念。

- 流
  已建立的连接上的双向字节流。

- 消息
  与逻辑消息对应的完整的一系列数据帧。

- 帧
  HTTP 2.0 通信的最小单位，每个帧包含帧首部，至少也会标识出当前帧所属的流。

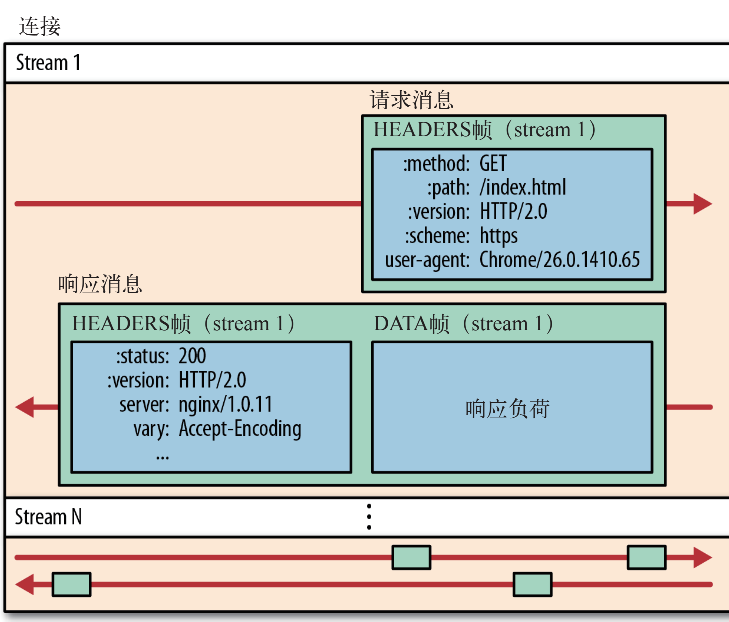


图 12-2：HTTP 2.0 流、消息和帧

所有 HTTP 2.0 通信都在一个连接上完成，这个连接可以承载任意数量的双向数据

流。相应地，每个数据流以消息的形式发送，而消息由一或多个帧组成，这些帧可以乱序发送，然后再根据每个帧首部的流标识符重新组装。

> HTTP 2.0 的所有帧都采用二进制编码，所有首部数据都会被压缩。因此，图 12-2 只是说明了数据流、消息和帧之间的关系，而非它们实际传输时的编码结果。要了解实际编码结果，请参考 12.4 节“二进制分帧简介”。

这简简单单的几句话里浓缩了大量的信息，我们再重申一次。要理解 HTTP 2.0，就必须理解流、消息和帧这几个基本概念。

- 所有通信都在一个 TCP 连接上完成。
- 流是连接中的一个虚拟信道，可以承载双向的消息；每个流都有一个唯一的整数标识符（1、2…*N*）。
- 消息是指逻辑上的 HTTP 消息，比如请求、响应等，由一或多个帧组成。
- 帧是最小的通信单位，承载着特定类型的数据，如 HTTP 首部、负荷，等等。

简言之，HTTP 2.0 把 HTTP 协议通信的基本单位缩小为一个一个的帧，这些帧对应着逻辑流中的消息。相应地，很多流可以并行地在同一个 TCP 连接上交换消息。

12.3.3　多向请求与响应

在 HTTP 1.x 中，如果客户端想发送多个并行的请求以及改进性能，那么必须使用多个 TCP 连接（参见 11.3 节“使用多个 TCP 连接”）。这是 HTTP 1.x 交付模型的直接结果，该模型会保证每个连接每次只交付一个响应（多个响应必须排队）。更糟糕的是，这种模型也会导致队首阻塞，从而造成底层 TCP 连接的效率低下。

HTTP 2.0 中新的二进制分帧层突破了这些限制，实现了多向请求和响应：客户端和服务器可以把 HTTP 消息分解为互不依赖的帧（图 12-3），然后乱序发送，最后再在另一端把它们重新组合起来。

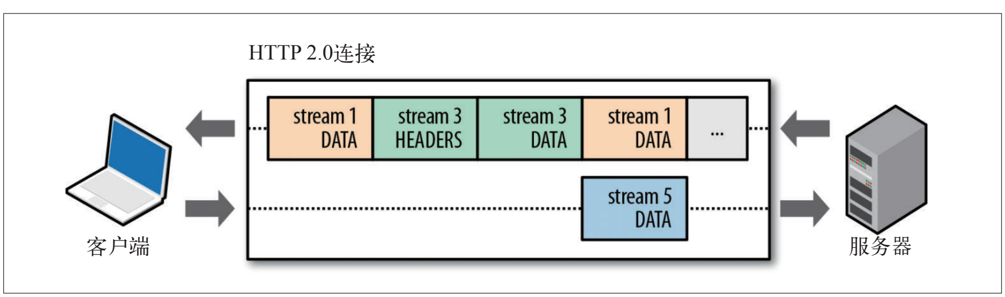


图 12-3：HTTP 2.0 在共享的连接上同时发送请求和响应

图 12-3 中包含了同一个连接上多个传输中的数据流：客户端正在向服务器传输一个 DATA 帧（stream 5），与此同时，服务器正向客户端乱序发送 stream 1 和 stream 3 的一系列帧。此时，一个连接上有 3 个请求 / 响应并行交换！

把 HTTP 消息分解为独立的帧，交错发送，然后在另一端重新组装是 HTTP 2.0 最重要的一项增强。事实上，这个机制会在整个 Web 技术栈中引发一系列连锁反应，从而带来巨大的性能提升，因为：

- 可以并行交错地发送请求，请求之间互不影响；
- 可以并行交错地发送响应，响应之间互不干扰；
- 只使用一个连接即可并行发送多个请求和响应；
- 消除不必要的延迟，从而减少页面加载的时间；
- 不必再为绕过 HTTP 1.x 限制而多做很多工作；
- 更多优势。

总之，HTTP 2.0 的二进制分帧机制解决了 HTTP 1.x 中存在的队首阻塞问题，也消除了并行处理和发送请求及响应时对多个连接的依赖。结果，就是应用速度更快、开发更简单、部署成本更低。

> 支持多向请求与响应，可以省掉针对 HTTP 1.x 限制所费的那些脑筋和工作，比如拼接文件、图片精灵、域名分区（参见 13.2 节“针对 HTTP 1.x 的优化建议”）。类似地，通过减少 TCP 连接的数量，HTTP 2.0 也会减少客户端和服务器的 CPU 及内存占用。

12.3.4　请求优先级

把 HTTP 消息分解为很多独立的帧之后，就可以通过优化这些帧的交错和传输顺序，进一步提升性能。为了做到这一点，每个流都可以带有一个 31 比特的优先值：

- 0 表示最高优先级；
- $2^{31}-1$ 表示最低优先级。

有了这个优先值，客户端和服务器就可以在处理不同的流时采取不同的策略，以最优的方式发送流、消息和帧。具体来讲，服务器可以根据流的优先级，控制资源分配（CPU、内存、带宽），而在响应数据准备好之后，优先将最高优先级的帧发送给客户端。

浏览器请求优先级与 HTTP 2.0

浏览器在渲染页面时，并非所有资源都具有相同的优先级：HTML 文档本身对构建 DOM 不可或缺，CSS 对构建 CSSOM 不可或缺，而 DOM 和 CSSOM 的构建都可能受到 JavaScript 资源的阻塞（参见 10.1 节的附注栏“DOM、CSSOM 和 JavaScript”），其他资源（如图片）的优先级都可以降低。

为加快页面加载速度，所有现代浏览器都会基于资源的类型以及它在页面中的位置排定请求的优先次序，甚至通过之前的访问来学习优先级模式——比如，之前的渲染如果被某些资源阻塞了，那么同样的资源在下一次访问时可能就会被赋予更高的优先级。

在 HTTP 1.x 中，浏览器极少能利用上述优先级信息，因为协议本身并不支持多路复用，也没有办法向服务器通告请求的优先级。此时，浏览器只能依赖并行连接，且最多只能同时向一个域名发送 6 个请求。于是，在等连接可用期间，请求只能在客户端排队，从而增加了不必要的网络延迟。理论上，HTTP 管道可以解决这个问题，只是由于缺乏支持而无法付诸实践。

HTTP 2.0 一举解决了所有这些低效的问题：浏览器可以在发现资源时立即分派请求，指定每个流的优先级，让服务器决定最优的响应次序。这样请求就不必排队了，既节省了时间，也最大限度地利用了每个连接。

HTTP 2.0 没有规定处理优先级的具体算法，只是提供了一种赋予数据优先级的机制，而且要求客户端与服务器必须能够交换这些数据。这样一来，优先值作为提示信息，对应的次序排定策略可能因客户端或服务器的实现而不同：客户端应该明确指定优先值，服务器应该根据该值处理和交付数据。

在这个规定之下，尽管你可能无法控制客户端发送的优先值，但或许你可以控制服务器。因此，在选择 HTTP 2.0 服务器时，可以多留点心！为说明这一点，考虑下面几个问题。

- 如果服务器对所有优先值视而不见怎么办？
- 高优先值的流一定优先处理吗？
- 是否存在不同优先级的流应该交错的情况？

如果服务器不理睬所有优先值，那么可能会导致应用响应变慢：浏览器明明在等关键的 CSS 和 JavaScript，服务器却在发送图片，从而造成渲染阻塞。不过，规定严格的优先级次序也可能带来次优的结果，因为这可能又会引入队首阻塞问题，即某个高优先级的慢请求会不必要地阻塞其他资源的交付。

服务器可以而且应该交错发送不同优先级别的帧。只要可能，高优先级流都应该优

先，包括分配处理资源和客户端与服务器间的带宽。不过，为了最高效地利用底层连接，不同优先级的混合也是必需的。

12.3.5　每个来源一个连接

有了新的分帧机制后，HTTP 2.0 不再依赖多个 TCP 连接去实现多流并行了。现在，每个数据流都拆分成很多帧，而这些帧可以交错，还可以分别优先级。于是，所有 HTTP 2.0 连接都是持久化的，而且客户端与服务器之间也只需要一个连接即可。

> 实验表明，客户端使用更少的连接肯定可以降低延迟时间。HTTP 2.0 发送的总分组数量比 HTTP 差不多要少 40%。而服务器处理大量并发连接的情况也变成了可伸缩性问题，因为 HTTP 2.0 减轻了这个负担。
>
> ——HT.TP/2.0
> Draft 2

每个来源一个连接显著减少了相关的资源占用：连接路径上的套接字管理工作量少了，内存占用少了，连接吞吐量大了。此外，从上到下所有层面上也都获得了相应的好处：

- 所有数据流的优先次序始终如一；
- 压缩上下文单一使得压缩效果更好；
- 由于 TCP 连接减少而使网络拥塞状况得以改观；
- 慢启动时间减少，拥塞和丢包恢复速度更快。

> 大多数 HTTP 连接的时间都很短，而且是突发性的，但 TCP 只在长时间连接传输大块数据时效率才最高。HTTP 2.0 通过让所有数据流共用同一个连接，可以更有效地使用 TCP 连接。

HTTP 2.0 不仅能够减少网络延迟，还有助于提高吞吐量和降低运营成本！

丢包、高 RTT 连接和 HTTP 2.0 性能

等一等，我听你说了一大堆每个来源一个 TCP 连接的好处，难道它就一点坏处都没有吗？有，当然有。

- 虽然消除了 HTTP 队首阻塞现象，但 TCP 层次上仍然存在队首阻塞（参见 2.4 节“队首阻塞”）；
- 如果 TCP 窗口缩放被禁用，那带宽延迟积效应可能会限制连接的吞吐量；
- 丢包时，TCP 拥塞窗口会缩小（参见 2.2.3 节“拥塞预防”）。

上述每一点都可能对 HTTP 2.0 连接的吞吐量和延迟性能造成不利影响。然而，除了这些局限性之外，实验表明一个 TCP 连接仍然是 HTTP 2.0 基础上的最佳部署策略：

> 目前为止的测试表明，压缩和优先级排定带来的性能提升，已经超过了队首阻塞（特别是丢包情况下）造成的负面效果。
>
> ——HTTP/2.0 Draft 2

与所有性能优化过程一样，去掉一个性能瓶颈，又会带来新的瓶颈。对 HTTP 2.0 而言，TCP 很可能就是下一个性能瓶颈。这也是为什么服务器端 TCP 配置对 HTTP 2.0 至关重要的一个原因。

目前，针对 TCP 性能优化的研究还在进行中：TCP 快速打开、比例降速、增大的初始拥塞窗口，等等不一而足。总之，一定要知道 HTTP 2.0 与之前的版本一样，并不强制使用 TCP。UDP 等其他传输协议也并非不可以。

12.3.6 流量控制

在同一个 TCP 连接上传输多个数据流，就意味着要共享带宽。标定数据流的优先级有助于按序交付，但只有优先级还不足以确定多个数据流或多个连接间的资源分配。为解决这个问题，HTTP 2.0 为数据流和连接的流量控制提供了一个简单的机制：

- 流量控制基于每一跳进行，而非端到端的控制；
- 流量控制基于窗口更新帧进行，即接收方广播自己准备接收某个数据流的多少字节，以及对整个连接要接收多少字节；
- 流量控制窗口大小通过 `WINDOW_UPDATE` 帧更新，这个字段指定了流 ID 和窗口大小递增值；
- 流量控制有方向性，即接收方可能根据自己的情况为每个流乃至整个连接设置任意窗口大小；
- 流量控制可以由接收方禁用，包括针对个别的流和针对整个连接。

> HTTP 2.0 连接建立之后，客户端与服务器交换 `SETTINGS` 帧，目的是设置双向的流量控制窗口大小。除此之外，任何一端都可以选择禁用个别流或整个连接的流量控制。

上面这个列表是不是让你想起了 TCP 流量控制？应该是，这两个机制实际上是一样的，参见 2.2.1 节“流量控制”。然而，由于 TCP 流量控制不能对同一条 HTTP 2.0 连接内的多个流实施差异化策略，因此光有它自己是不够的。这正是 HTTP 2.0 流

量控制机制出台的原因。

HTTP 2.0 标准没有规定任何特定的算法、值，或者什么时候发送 WINDOW_UPDATE 帧。因此，实现可以选择自己的算法以匹配自己的应用场景，从而求得最佳性能。

> 优先级可以决定交付次序，而流量控制则可以控制 HTTP 2.0 连接中每个流占用的资源：接收方可以针对特定的流广播较低的窗口大小，以限制它的传输速度。

12.3.7 服务器推送

HTTP 2.0 新增的一个强大的新功能，就是服务器可以对一个客户端请求发送多个响应。换句话说，除了对最初请求的响应外，服务器还可以额外向客户端推送资源（图 12-4），而无需客户端明确地请求。

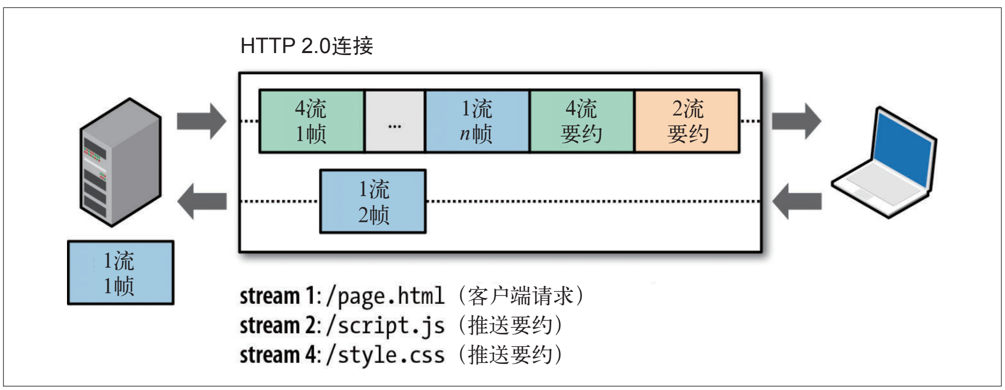


图 12-4：服务器发起推送资源的新流（要约）

> 建立 HTTP 2.0 连接后，客户端与服务器交换 SETTINGS 帧，借此可以限定双向并发的流的最大数量。因此，客户端可以限定推送流的数量，或者通过把这个值设置为 0 而完全禁用服务器推送。

为什么需要这样一个机制呢？通常的 Web 应用都由几十个资源组成，客户端需要分析服务器提供的文档才能逐个找到它们。那为什么不让服务器提前就把这些资源推送给客户端，从而减少额外的时间延迟呢？服务器已经知道客户端下一步要请求什么资源了，这时候服务器推送即可派上用场。事实上，如果你在网页里嵌入过 CSS、JavaScript，或者通过数据 URI 嵌入过其他资源（参见 11.7 节“嵌入资源”），那你就已经亲身体验过服务器推送了。

把资源直接插入到文档中，就是把资源直接推送给客户端，而无需客户端请求。在 HTTP 2.0 中，唯一的不同就是可以把这个过程从应用中拿出来，放到 HTTP 协议本身来实现，而且还带来了如下好处：

- 客户端可以缓存推送过来的资源；
- 客户端可以拒绝推送过来的资源；
- 推送资源可以由不同的页面共享；
- 服务器可以按照优先级推送资源。

> 所有推送的资源都遵守同源策略。换句话说，服务器不能随便将第三方资源推送给客户端，而必须是经过双方确认才行。

有了服务器推送后，HTTP 1.x 时代的大多数插入或嵌入资源的做法基本上也就过时了。唯一有必要直接在网页中插入资源的情况，就是该资源只供那一个网页使用，而且编码代价不大；此处仍然可以参考 11.7 节“嵌入资源”。除此之外，所有应用都应该使用 HTTP 2.0 服务器推送。

> **PUSH_PROMISE**
>
> 所有服务器推送流都由 PUSH_PROMISE 发端，它是除了对原始请求的响应之外，服务器向客户端发出的有意推送所述资源的信号。PUSH_PROMISE 帧中只包含要约（promise）资源的 HTTP 首部。
>
> 客户端接收到 PUSH_PROMISE 帧之后，可以视自身需求选择拒绝这个流（比如，已经缓存了相应资源），而这是对 HTTP 1.x 的一个重要改进。嵌入资源作为针对 HTTP 1.x 的一种流行“优化技巧”，实际上无异于“强制推送”：客户端无法取消这种“推送”，而且也不能个别地缓存嵌入的资源。
>
> 最后再说一说服务器推送的几点限制。首先，服务器必须遵循请求 – 响应的循环，只能借着对请求的响应推送资源。也就是说，服务器不能随意发起推送流。其次，PUSH_PROMISE 帧必须在返回响应之前发送，以免客户端出现竞态条件。否则，就可能出现比如这种情况：客户端请求的恰好是服务器打算推送的资源。

实现HTTP 2.0服务器推送

服务器推送为优化应用的资源交付提供了很多可能。然而，服务器到底如何确定哪些资源可以或应该推送呢？与确定优先级类似，HTTP 2.0 标准也没有就此规定某种算法，所以实现者就拥有了解释权。自然地，也就有可能出现多种策略，每种策略可能会考虑一种应用或服务器使用场景。

- 应用可以在自身的代码中明确发起服务器推送。这种情况要求与 HTTP 2.0 紧密耦合，但开发人员有控制权。
- 应用可以通过额外的 HTTP 首部向服务器发送信号，列出它希望推送的资源。这样可以将应用与 HTTP 服务器 API 分离。比如 Apache 的 mod_spdy 能够识别 `X-Associated-Content` 首部，这个首部中列出了希望服务器推送的资源。
- 服务器可以不依赖应用而自动学习相关资源。服务器可以解析文档，推断出要推送的资源，或者可以分析流量，然后作出适当的决定。比如服务器可以根据 `Referer` 首部收集依赖数据，然后自动向客户端推送关键资源。

当然，以上只是各种可能策略中的几个，但由此也可以知道可能性是很多的：可能是手工调用低级 API，也可能是一种全自动的实现。类似地，服务器应不应该重复推送相同的资源，还是应该实现一个更智能的策略？服务器可以根据自身的模型、客户端 cookie 或其他机制，智能推断出客户端缓存中有什么资源，然后再作出推送决定。简言之，服务器推送领域将爆出各种创新。

最后还有一点，就是推送的资源将直接进入客户端缓存，就像客户端请求了似的。不存在客户端 API 或 JavaScript 回调方法等通知机制，可以用于确定资源何时到达。整个过程对运行在浏览器中的 Web 应用来说好像根本不存在。

12.3.8 首部压缩

HTTP 的每一次通信都会携带一组首部，用于描述传输的资源及其属性。在 HTTP 1.x 中，这些元数据都是以纯文本形式发送的，通常会给每个请求增加 500~800 字节的负荷。如果算上 HTTP cookie，增加的负荷通常会达到上千字节（参见 11.5 节“度量和控制协议开销”）。为减少这些开销并提升性能，HTTP 2.0 会压缩首部元数据：

- HTTP 2.0 在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键－值对，对于相同的数据，不再通过每次请求和响应发送；
- 首部表在 HTTP 2.0 的连接存续期内始终存在，由客户端和服务器共同渐进地更新；
- 每个新的首部键－值对要么被追加到当前表的末尾，要么替换表中之前的值。

于是，HTTP 2.0 连接的两端都知道已经发送了哪些首部，这些首部的值是什么，从而可以针对之前的数据只编码发送差异数据（图 12-5）。

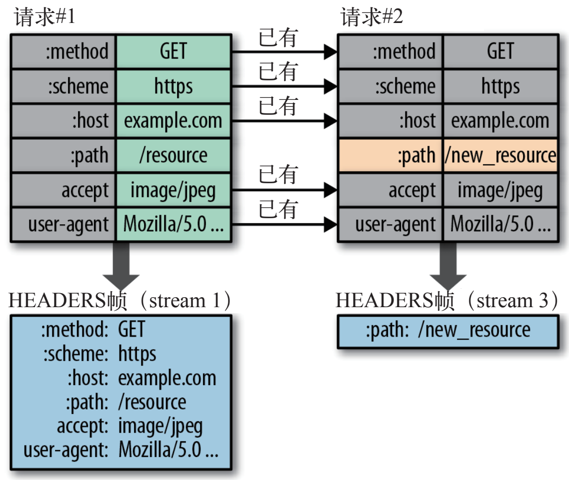


图 12-5：HTTP 2.0 首部的差异化编码

> 请求与响应首部的定义在 HTTP 2.0 中基本没有改变，只是所有首部键必须全部小写，而且请求行要独立为 `:method`、`:scheme`、`:host` 和 `:path` 这些键－值对。

在前面的例子中，第二个请求只需要发送变化了的路径首部（`:path`），其他首部没有变化，不用再发送了。这样就可以避免传输冗余的首部，从而显著减少每个请求的开销。

通信期间几乎不会改变的通用键－值对（用户代理、可接受的媒体类型，等等）只需发送一次。事实上，如果请求中不包含首部（例如对同一资源的轮询请求），那么首部开销就是零字节。此时所有首部都自动使用之前请求发送的首部！

SPDY、CRIME 和 HTTP 2.0 压缩

SPDY 的早期版本使用 zlib 和自定义的字典压缩所有 HTTP 首部，可以减少 85%~88% 的首部开销，从而显著减少加载页面的时间：

> 在低速 DSL 连接中，上传速度只有 375 Kbit/s，仅压缩请求首部，即可显著减少某些（需要发送大量资源请求的）站点的页面加载时间。我们发现压缩首部可以节省 45~1142 ms 的页面加载时间。
>
> ——SPDY 白皮书，chromium.org

然而，2012 年夏天，出现了针对 TLS 和 SPDY 压缩算法的“CRIME”安全攻击，它可以利用首部压缩拦截会话。于是，zlib 压缩算法被撤销，取而代之的是前面介绍的新索引表算法。索引表算法没有类似的安全问题，但可以实现相差无几的性能提升。

要全面了解 HTTP 2.0 压缩算法，请看这里：http://tools.ietf.org/html/draft-ietf-httpbis-header-compression。

12.3.9　有效的HTTP 2.0升级与发现

向 HTTP 2.0 的迁移不可能一蹴而就。数百万台服务器必须升级才能使用新的二进制分帧，还有数十亿客户端必须更新浏览器和网络库。

好消息是，大多数现代浏览器都内置有高效的后台升级机制，因而对相当大一部分既有用户来说，这些浏览器能很快支持 HTTP 2.0，又不会带来太多干扰。尽管如此，还是有一部分用户只能使用旧版本的浏览器，而服务器和中间设备也必须升级支持 HTTP 2.0，这需要一个比较长的时期，而且是一个费力、费钱的过程。

HTTP 1.x 至少还会存在十年以上，大多数服务器和客户端在此期间必须同时支持 1.x 和 2.0 标准。于是，支持 HTTP 2.0 的客户端在发起新请求之前，必须能发现服务器及所有中间设备是否支持 HTTP 2.0 协议。有三种可能的情况：

- 通过 TLS 和 ALPN 发起新的 HTTPS 连接；
- 根据之前的信息发起新的 HTTP 连接；
- 没有之前的信息而发起新的 HTTP 连接。

HTTPS 协商过程中有一个环节会使用 ALPN（Application Layer Protocol Negotiation）发现和协商 HTTP 2.0 的支持情况 [参见 4.2.1 节“应用层协议协商（ALPN）”]。减少网络延迟是 HTTP 2.0 的关键条件，因此在建立 HTTPS 连接时一定会用到 ALPN 协商。

通过常规非加密信道建立 HTTP 2.0 连接需要多做一点工作。因为 HTTP 1.0 和 HTTP 2.0 都使用同一个端口（80），又没有服务器是否支持 HTTP 2.0 的其他任何信息，此时客户端只能使用 HTTP `Upgrade` 机制通过协调确定适当的协议：

```
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: HTTP/2.0 ❶
HTTP2-Settings: (SETTINGS payload) ❷
```

```
HTTP/1.1 200 OK ❸
Content-length: 243
Content-type: text/html

(... HTTP 1.1 response ...)

          (or)

HTTP/1.1 101 Switching Protocols ❹
Connection: Upgrade
Upgrade: HTTP/2.0

(... HTTP 2.0 response ...)
```

❶ 发起带有 HTTP 2.0 Upgrade 首部的 HTTP 1.1 请求
❷ HTTP/2.0 `SETTINGS` 净荷的 Base64 URL 编码
❸ 服务器拒绝升级，通过 HTTP 1.1 返回响应
❹ 服务器接受 HTTP 2.0 升级，切换到新分帧

使用这种 `Upgrade` 流，如果服务器不支持 HTTP 2.0，就立即返回 HTTP 1.1 响应。否则，服务器就会以 HTTP 1.1 格式返回 `101 Switching Protocols` 响应，然后立即切换到 HTTP 2.0 并使用新的二进制分帧协议返回响应。无论哪种情况，都不需要额外往返。

> 为确定服务器和客户端都有意使用 HTTP 2.0 对话，双方还必须发送“连接首部”，也就是一串标准的字节。这种信息交换本质上是一种“尽早失败”（fail-fast）的机制，可以避免客户端、服务器，以及中间设备偶尔接受请求的升级却不理解新协议。而且，这种信息交换也不会带来额外的往返，只是在连接开始时要多传一些字节。

最后，如果客户端因为自己保存有或通过其他手段（如 DNS 记录、手工配置等）获得了关于 HTTP 2.0 的支持信息，它也可以直接发送 HTTP 2.0 分帧，而不必依赖 `Upgrade` 机制。有了这些信息，客户端可以一上来就通过非加密信道发送 HTTP 2.0 分帧，其他就不管了。最坏的情况，就是无法建立连接，客户端再回退一步，重新使用 `Upgrade` 首部，或者切换到带 ALPN 协商的 TLS 信道。

部署 HTTP 2.0 的同时部署 TLS 和 ALPN

服务器之前的 HTTP 2.0 支持信息并不能保证下一次就能可靠地建立连接。以这种方式通信的前提，就是各端都必须支持 HTTP 2.0。如果任何中间设备不支持，连接都不会成功。

因此，尽管 HTTP 2.0 并不要求使用 TLS，但在实践中，将其部署到现有的大量中间设备仍然是最可靠的策略（参见 4.1 节中的“Web 代理、中间设备、TLS 与新协议”）。最好的结果，就是除了常规的 `Upgrade` 工作流，还要在部署 HTTP 2.0 时，一起部署带 ALPN 协商的 TLS。

12.4 二进制分帧简介

HTTP 2.0 的根本改进还是新增的长度前置的二进制分帧层。与 HTTP 1.x 使用换行符分隔纯文本不同，二进制分帧层更加简洁，通过代码处理起来更简单也更有效。

建立了 HTTP 2.0 连接后，客户端与服务器会通过交换帧来通信，帧是基于这个新协议通信的最小单位。所有帧都共享一个 8 字节的首部（图 12-6），其中包含帧的长度、类型、标志，还有一个保留位和一个 31 位的流标识符。

Bit	+0..7	+8..15	+16..23	+24..31
0	长度		类型	标志
32	R 流标识符			
...	帧净荷			

图 12-6：共有的 8 字节帧首部

- 16 位的长度前缀意味着一帧大约可以携带 64 KB 数据，不包括 8 字节首部。
- 8 位的类型字段决定如何解释帧其余部分的内容。
- 8 位的标志字段允许不同的帧类型定义特定于帧的消息标志。
- 1 位的保留字段始终置为 0。
- 31 位的流标识符唯一标识 HTTP 2.0 的流。

在调试 HTTP 2.0 通信时，有人会使用自己喜欢的十六进制查看器。其实，Wireshark 及其他类似的工具也有相应的插件，使用很简单，也很人性化。比如，谷歌 Chrome 就支持 `chrome://internals#spdy`，通过它可以查看通信细节。

知道了 HTTP 2.0 规定的这个共享的帧首部，就可以自己编写一个简单的解析器，通过分析 HTTP 2.0 字节流，根据每个帧的前 8 字节找到帧的类型、标志和长度。而且，由于每个帧的长度都是预先定义好的，解析器可以迅速而准确地跳到下一帧的开始，这也是相对于 HTTP 1.x 的一个很大的性能提升。

知道了帧类型，解析器就知道该如何解释帧的其余内容了。HTTP 2.0 规定了如下帧类型。

- `DATA`：用于传输 HTTP 消息体。
- `HEADERS`：用于传输关于流的额外的首部字段。
- `PRIORITY`：用于指定或重新指定引用资源的优先级。
- `RST_STREAM`：用于通知流的非正常终止。
- `SETTINGS`：用于通知两端通信方式的配置数据。
- `PUSH_PROMISE`：用于发出创建流和服务器引用资源的要约。
- `PING`：用于计算往返时间，执行"活性"检查。
- `GOAWAY`：用于通知对端停止在当前连接中创建流。
- `WINDOW_UPDATE`：用于针对个别流或个别连接实现流量控制。
- `CONTINUATION`：用于继续一系列首部块片段。

> 服务器可以利用 `GOAWAY` 类型的帧告诉客户端要处理的最后一个流的 ID，从而消除一些请求竞争，而且浏览器也可以据此智能地重试或取消"悬着的"请求。这也是保证复用连接安全的一个重要和必要的功能！

前述各种类型帧的具体实现在很大程度上取决于服务器和客户端开发商，他们需要考虑流量控制、错误处理、连接终止等细节。好在，所有这些内容在官方标准中都有论述。好奇的话，可以查阅一下最新的草案。

既然有了这个分帧层，即使它对我们的应用不可见，我们也应该更进一步，分析一下两种最常见的工作流：发起新流和交换应用数据。只有明白了一个请求或响应如何转换成一个一个的帧，才能理解 HTTP 2.0 对性能的提升来自哪里。

固定长度与可变长度字段

HTTP 2.0 只使用固定长度字段，HTTP 2.0 帧占用带宽很少（帧首部是 8 字节）。采用可变长度编码的确可以节省一点带宽和时延，但却无法抵偿由此带来的分析复杂性。

即使可变长度编码能减少 50% 的带宽占用，那么在 1 Mbit/s 的连接上传输 1400 字节的分组，也只能节省 4 字节（0.3%）和每帧不到 100 纳秒的延迟时间。

12.4.1 发起新流

在发送应用数据之前，必须创建一个新流并随之发送相应的元数据，比如流优先级、HTTP 首部等。HTTP 2.0 协议规定客户端和服务器都可以发起新流，因此有两种可能：

- 客户端通过发送 `HEADERS` 帧来发起新流（图 12-7），这个帧里包含带有新流 ID 的公用首部、可选的 31 位优先值，以及一组 HTTP 键－值对首部；
- 服务器通过发送 `PUSH_PROMISE` 帧来发起推送流，这个帧与 `HEADERS` 帧等效，但它包含“要约流 ID”，没有优先值。

Bit	+0..7	+8..15	+16..23	+24..31
0	长度		类型	标志
32	R 流标识符			
64	R 优先值			
...	首部块			

图 12-7：带优先值的 HEADERS 帧

这两种帧的类型字段都只用于沟通新流的元数据，净荷会在 `DATA` 帧中单独发送。同样，由于两端都可以发起新流，流计数器偏置：客户端发起的流具有偶数 ID，服务器发起的流具有奇数 ID。这样，两端的流 ID 不会冲突，而且各自持有一个简单的计数器，每次发起新流时递增 ID 即可。

> 由于流的元数据与应用数据是单独发送的，因此客户端和服务器可以分别给它们设定不同的优先级。比如，“控制流量”的流优先级可以高一些，但只将其应用给 `DATA` 帧。

12.4.2 发送应用数据

创建新流并发送 HTTP 首部之后，接下来就是利用 `DATA` 帧（图 12-8）发送应用数据。应用数据可以分为多个 `DATA` 帧，最后一帧要翻转帧首部的 `END_STREAM` 字段。

Bit	+0..7	+8..15	+16..23	+24..31
0	长度		类型	标志
32	R 流标识符			
...	帧净荷			

图 12-8：DATA 帧

数据净荷不会被另行编码或压缩。编码方式取决于应用或服务器，纯文本、gzip 压缩、图片或视频压缩格式都可以。既然如此，关于 `DATA` 帧再也没有什么新东西好说了！整个帧由公用的 8 字节首部，后跟 HTTP 净荷组成。

从技术上说，DATA 帧的长度字段决定了每帧的数据净荷最多可达 $2^{16}-1$（65 535）字节。可是，为减少队首阻塞，HTTP 2.0 标准要求 DATA 帧不能超过 $2^{14}-1$（16 383）字节。长度超过这个阀值的数据，就得分帧发送。

12.4.3　HTTP 2.0帧数据流分析

了解了不同帧的基础知识，下面我们再看一看 12.3.3 节“多向请求与响应”中的那张图（图 12-9），分析一下数据流。

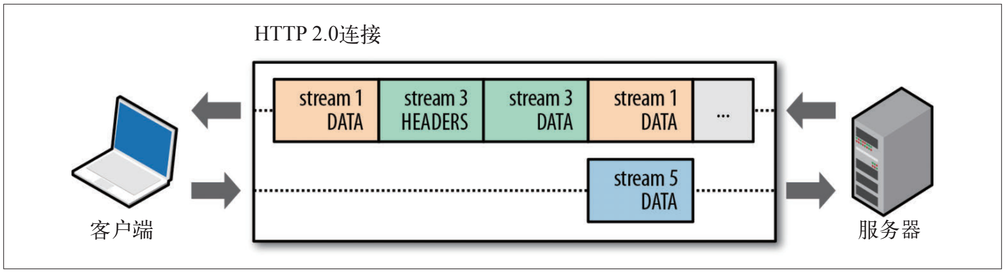


图 12-9：HTTP 2.0 在共享的连接上同时发送请求和响应

- 有 3 个活动的流：stream 1、stream 3 和 stream 5。
- 3 个流的 ID 都是奇数，说明都是客户端发起的。
- 这里没有服务器发起的流。
- 服务器发送的 stream 1 包含多个 DATA 帧，这是对客户端之前请求的响应数据。这也说明在此之前已经发送过 HEADERS 帧了。
- 服务器在交错发送 stream 1 的 DATA 帧和 stream 3 的 HEADERS 帧，这就是响应的多路复用！
- 客户端正在发送 stream 5 的 DATA 帧，表明 HEADERS 帧之前已经发送过了。

简言之，图 12-9 中连接正在并行传送 3 个数据流，每个流都处于各自处理周期的不同阶段。服务器决定帧的顺序，而我们不用关心每个流的类型或内容。stream 1 携带的数据量可能比较大，也许是视频，但它不会阻塞共享连接中的其他流！

第 13 章

优化应用的交付

高性能浏览器网络有赖于大量网络技术（图 13-1），而我们应用的整体性能则是所有这些组成部分性能表现之和。

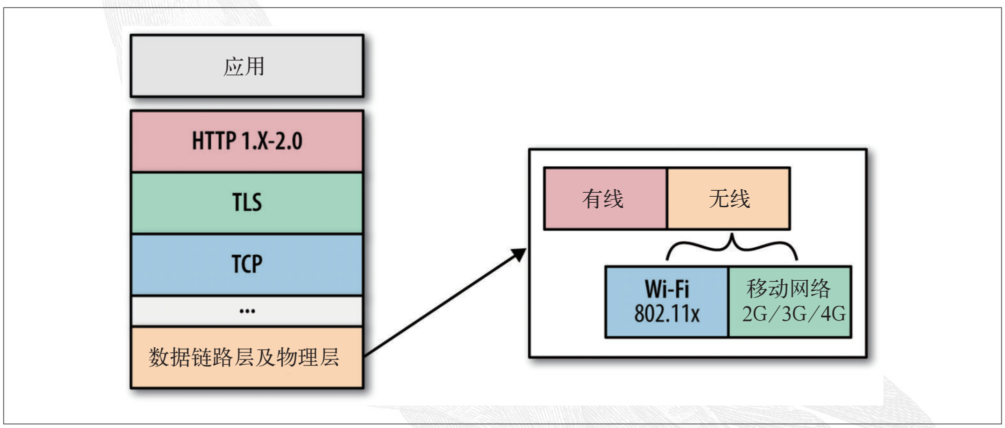


图 13-1：与优化应用交付相关的所有层

我们无法控制客户端与服务器之间的网络环境，也不能控制客户的硬件或者其手持设备的配置，但除此之外的一切就掌握在我们手里了，包括服务器上的 TCP 和 TLS 优化，以及针对不同物理层特性、不同 HTTP 协议版本和通用最佳实践的数十项应用优化。没错，要做到滴水不漏绝非易事，但这样做绝对值得！好，下面我们就来作一番综述。

通信信道的物理属性对所有应用而言都是一项硬性限制：光速及客户端与服务器之间的距离决定了信号传播的延迟，而媒介（有线或无线）决定了由每个数据分组带来的处理、传输、排队及其他延迟。事实上，影响绝大多数 Web 应用性能的并非带宽，而是延迟。网速虽然越来越快，但不幸的是，延迟似乎并没有缩短：

- 1.2 节“延迟的构成”；
- 1.7 节“目标：高带宽和低延迟”；
- 10.3.2 节“延迟是性能瓶颈”。

既然我们不能让信息跑得更快，那么关键就在于对传输层和应用层采取各种可能的优化手段，消除不必要的往返、请求，把每个分组的传输距离缩到最短——比如把服务器放到离客户更近的地方。

针对无线网络物理层的特有属性采取优化措施可以让任何应用受益，因为无线环境的延迟高且带宽总是那么贵。对 API 而言，有线与无线网络之间的差别特别明显，对此视而不见可不明智。只要对何时以及如何下载资源、信标进行简单的优化，就能显著改善用户感觉到的延迟、电池使用时间和应用的整体用户体验：

- 6.4 节“针对 Wi-Fi 的优化建议”；
- 第 8 章“移动网络的优化建议”。

自物理层向上，接下来就是要保证任何一台服务器都要按照最新的 TCP 和 TLS 最佳实践进行配置。针对底层协议的优化能保证每个客户端在与服务器通信时，都可以获取最佳性能——高吞吐量和低延迟：

- 2.5 节“针对 TCP 的优化建议”；
- 4.7 节“针对 TLS 的优化建议”。

最后，就是应用层。无论从哪个角度讲，HTTP 都是出奇成功的一个协议。毕竟，它是数十亿客户端与服务器交流的“通用语言”，没有它就没有现代 Web（万维网）。可是，HTTP 也是一个不完美的协议，这就意味着我们在架构自己的应用时必须格外小心：

- 我们必须想方设法地绕过 HTTP 1.x 的种种限制；
- 我们必须掌握利用 HTTP 2.0 性能增强的方法；
- 我们必须在应用经典的性能最佳实践时保持警惕。

说到底，成功的、可持续的 Web 性能优化策略其实很简单：先度量，然后拿业务目标与性能指标进行比较，采取优化措施，紧了松点，松了紧点，如此反复。开发和购买合用的度量工具及选择恰当的度量手段具有最高优先级；参见 10.4 节“人造和真实用户性能度量”。

13.1 经典的性能优化最佳实践

无论什么网络，也不管所用网络协议是什么版本，所有应用都应该致力于消除或减少不必要的网络延迟，将需要传输的数据压缩至最少。这两条标准是经典的性能优化最佳实践，是其他数十条性能准则的出发点。

- *减少DNS查找*
  每一次主机名解析都需要一次网络往返，从而增加请求的延迟时间，同时还会阻塞后续请求。

- *重用TCP连接*
  尽可能使用持久连接，以消除 TCP 握手和慢启动延迟；参见 2.2.2 节“慢启动”。

- *减少HTTP重定向*
  HTTP 重定向极费时间，特别是不同域名之间的重定向，更加费时；这里面既有额外的 DNS 查询、TCP 握手，还有其他延迟。最佳的重定向次数为零。

- *使用CDN（内容分发网络）*
  把数据放到离用户地理位置更近的地方，可以显著减少每次 TCP 连接的网络延迟，增大吞吐量。这一条既适用于静态内容，也适用于动态内容；参见 4.7.2 节中的“不缓存的原始获取”。

- *去掉不必要的资源*
  任何请求都不如没有请求快。

说到这，所有建议都无需解释。延迟是瓶颈，最快的速度莫过于什么也不传输。然而，HTTP 也提供了很多额外的机制，比如缓存和压缩，还有与其版本对应的一些性能技巧。

- *在客户端缓存资源*
  应该缓存应用资源，从而避免每次请求都发送相同的内容。

- *传输压缩过的内容*
  传输前应该压缩应用资源，把要传输的字节减至最少：确保对每种要传输的资源采用最好的压缩手段。

- 消除不必要的请求开销

  减少请求的 HTTP 首部数据（比如 HTTP cookie），节省的时间相当于几次往返的延迟时间。

- 并行处理请求和响应

  请求和响应的排队都会导致延迟，无论是客户端还是服务器端。这一点经常被忽视，但却会无谓地导致很长延迟。

- 针对协议版本采取优化措施

  HTTP 1.x 支持有限的并行机制，要求打包资源、跨域分散资源，等等。相对而言，HTTP 2.0 只要建立一个连接就能实现最优性能，同时无需针对 HTTP 1.x 的那些优化方法。

上述所有各项均有必要详细解释。下面我们分别讨论。

13.1.1　在客户端缓存资源

要说最快的网络请求，那就是不用发送请求就能获取资源。将之前下载过的数据缓存并维护好，就可以做到这一点。对于通过 HTTP 传输的资源，要保证首部包含适当的缓存字段：

- `Cache-Control` 首部用于指定缓存时间；
- `Last-Modified` 和 `ETag` 首部提供验证机制。

只要可能，就给每种资源都指定一个明确的缓存时间。这样客户端就可以直接使用本地副本，而不必每次都请求相同的内容。类似地，指定验证机制可以让客户端检查过期的资源是否有更新。没有更新，就没必要重新发送。

最后，还要注意应同时指定缓存时间和验证方法！只指定其中之一是最常见的错误，于是要么导致每次都在没有更新的情况下重发相同内容（这是没有指定验证），要么导致每次使用资源时都多余地执行验证检查（这是没有指定缓存时间）。

理想与现实：智能手机上的 Web 资源缓存

对 HTTP 最早的版本而言，缓存 HTTP 资源一度是最最重要的性能优化措施。然而，尽管看似所有人都意识到了这一措施的好处，但实际调查却不断发现，缓存资源却经常是一个被忽略的措施！ AT&T 实验室与密歇根大学最近的一次合作研究指出：

> 我们对两个数据集的调查表明，冗余的传输分别占据相应 HTTP 总流量的 18% 和 20%。这些额外传输的内容占总字节数的 17%，耗电 7%，占信号负载的 6%，在第二个数据集的所有蜂窝数据流量中，占无线资源的 9%。而产生这些冗余传输的主要原因，可以归结为智能手机 Web 缓存的实现未能完全支持或者严格遵照协议规范，或者开发人员没有完全利用相应的库所提供的缓存功能。
>
> ——Web Caching on Smartphones，*MobiSys* 2012

你的应用在一遍又一遍地请求不必要的资源吗？有证据表明，提出这个问题一点也不可笑。请大家务必对自己的应用多加注意，当然最好是通过测试手段来保证。

13.1.2　压缩传输的数据

利用本地缓存可以让客户端避免每次请求都重复取得数据。不过，还是有一些资源是必须取得的，比如原来的资源过期了，或者有新资源，再或者资源不能缓存。对于这些资源，应该保证传输的字节数最少。因此要保证对它们进行最有效的压缩。

HTML、CSS 和 JavaScript 等文本资源的大小经过 gzip 压缩平均可以减少 60%~80%。而图片则需要仔细考量：

- 图片一般会占到一个网页需要传输的总字节数的一半；
- 通过去掉不必要的元数据可以把图片文件变小；
- 要调整大小就在服务器上调整，避免传输不必要的字节；
- 应该根据图像选择最优的图片格式；
- 尽可能使用有损压缩。

不同图片格式的压缩率迥然不同，因为不同的格式是分别为不同使用场景设计的。事实上，如果选错了图片格式（比如，使用了 PNG 而非 JPG 或 WebP），多产生几百甚至上千 KB 数据是轻而易举的事。建议大家多找一些工具和自动化手段，以确定最佳图片格式。

选定图片格式后，其次就是不要让图片超过它需要的大小。如果在客户端对超出需要大小的图片做调整，那么除了额外传输不必要的字节之外，还会浪费 CPU、GPU 和内存资源（参见 11.6 节的“计算图片对内存的需求”）。

最后，选择了正确的格式，确定了必需的大小，接下来就要研究使用哪一种有损图片格式，比如 JPEG 还是 WebP，以及压缩到哪个级别：较高压缩率可以明显减少字节数，同时图片品质不会有太大或太明显的损失，尤其是在较小（手机）的屏幕上看，不容易发现。

> **WebP：Web 上的新图片格式**
>
> WebP 是谷歌开发的一种新图片格式，得到了 Chrome 和 Opera 浏览器支持。这种格式的无损压缩和有损压缩效能都有所提升：
>
> - WebP 的无损压缩图片比 PNG 的小 26%；
> - WebP 的有损压缩图片比 JPG 的小 25%~34%；
> - WebP 支持无损透明压缩，但因此仅增加 22% 的字节。
>
> 在现有网页平均 1 MB 大小，其中图片占一半的情况下，WebP 节省的 20%~30%，对每个页面而言就是几百 KB。这种格式需要客户端 CPU 多花点时间解码（大约相当于处理 JPG 的 1.4 倍），但字节的节省完全可以补偿处理时间的增长。此外，由于数据流量的限制和高速网络的存在，对很多用户而言，节省字节才是当务之急。
>
> 事实上，Chrome Data Compression Proxy 和 Opera Turbo 等工具为用户降低带宽占用的主要手段，就是重新把每张图片编码为 WebP 格式。正常情况下，Chrome Data Compression Proxy 的数据压缩率可以达到 50%，这说明我们自己的应用也有很多可以通过压缩提升性能的空间。

13.1.3　消除不必要的请求字节

HTTP 是一种无状态协议，也就是说服务器不必保存每次请求的客户端的信息。然而，很多应用又依赖于状态信息以实现会话管理、个性化、分析等功能。为了实现这些功能，HTTP State Management Mechanism（RFC 2965）作为扩展，允许任何网站针对自身来源关联和更新 cookie 元数据：浏览器保存数据，而在随后发送给来源的每一个请求的 `Cookie` 首部中自动附加这些信息。

上述标准并未规定 cookie 最大不能超过多大，但实践中大多数浏览器都将其限制为 4 KB。与此同时，该标准还规定每个站点针对其来源可以有多个关联的 cookie。于是，一个来源的 cookie 就有可能多达几十 KB！不用说，这么多元数据随请求传递，必然会给应用带来明显的性能损失：

- 浏览器会在每个请求中自动附加关联的 cookie 数据；
- 在 HTTP 1.x 中，包括 cookie 在内的所有 HTTP 首部都会在不压缩的状态下传输；
- 在 HTTP 2.0 中，这些元数据经过压缩了，但开销依然不小；
- 最坏的情况下，过大的 HTTP cookie 会超过初始的 TCP 拥塞窗口，从而导致多余的网络往返。

应该认真对待和监控 cookie 的大小，确保只传输最低数量的元数据，比如安全会话令牌。同时，还应该利用服务器上共享的会话缓存，从中查询缓存的元数据。更好

的结果，则是完全不用 cookie。比如，在请求图片、脚本和样式表等静态资源时，浏览器绝大多数情况下不必传输特定于客户端的元数据。

> 在使用 HTTP 1.x 的情况下，可以指定一个专门的“无需 cookie”的来源服务器。这个服务器可以用于交付那些不区分客户端的共用资源。

13.1.4 并行处理请求和响应

为了让应用响应速度达到最快，应该尽可能第一时间就分派所有资源请求。可是，还有一点也要考虑到，那就是所有这些请求以及它们对应的响应，将会被服务器如何处理。如果我们的请求在服务器上按先来后到的顺序依次排队，那就又会导致不必要的延迟。要是想实现最佳性能，就要记住以下几点：

- 使用持久连接，从 HTTP 1.0 升级到 HTTP 1.1；
- 利用多个 HTTP 1.1 连接实现并行下载；
- 可能的情况下利用 HTTP 1.1 管道；
- 考虑升级到 HTTP 2.0 以提升性能；
- 确保服务器有足够的资源并行处理请求。

如果不使用持久连接，则每个 HTTP 请求都要建立一个 TCP 连接。由于 TCP 握手和慢启动，多个 TCP 会造成明显的延迟。在使用 HTTP 1.1 的情况下，最好尽可能重用已有连接。如果碰上能使用 HTTP 管道的机会，不要放过。更好的选择，则是升级到 HTTP 2.0，从而获得最佳性能。

识别造成客户端和服务器延迟的不必要资源既是艺术也是技术：要仔细检查客户端资源瀑布（参见 10.2.2 节“分析资源瀑布”），以及服务器日志。常见问题如下：

- 服务器资源不足，造成不必要的资源处理延迟；
- 代理及负载均衡器容量不足，造成向应用服务器交付请求的延迟（请求排队）；
- 客户端资源阻塞导致页面构建延迟，参见 10.1 节的“DOM、CSSOM 和 JavaScript”。

优化浏览器中的资源加载

浏览器会自动确定文档中每个资源的最优加载顺序，但我们可以为浏览器提供辅助，也可以给它帮倒忙：

- 可以为浏览器给出提示，参见 10.5 节“针对浏览器的优化建议”；
- 可以通过藏匿资源来干扰浏览器。

现代浏览器都会尽可能高效迅速地扫描 HTML 和 CSS 文件内容。可是，文档解析器也会因为下载脚本或其他阻塞资源而被迫等待。此时，浏览器会使用“预加载扫描器”，推测有哪些资源要下载，从而尽早分派请求，以减少总体延迟。

但使用预加载扫描器是一种推测性优化，而且只会在文档解析器被阻塞的情况下才会使用。实践表明，这种推测性优化效果很好：根据谷歌 Chrome 的试验性数据，这种优化能把页面加载时间和渲染速度提高约 20%！

可惜的是，这种优化不适合通过 JavaScript 调度的资源。毕竟预加载扫描器不能预先执行脚本啊。结果，通过脚本来调度资源虽然可以更细化地控制应用，但也向预加载扫描器藏匿了资源，这种做法的得失还要仔细地权衡。

13.2 针对HTTP 1.x的优化建议

针对 HTTP 1.x 的优化次序很重要：首先要配置服务器以最大限度地保证 TCP 和 TLS 的性能最优，然后再谨慎地选择和采用移动及经典的应用最佳实践，之后再度量，迭代。

采用了经典的应用优化措施和适当的性能度量手段，还要进一步评估是否有必要为应用采取特定于 HTTP 1.x 的优化措施（其实是权宜之计）。

- 利用HTTP管道

  如果你的应用可以控制客户端和服务器这两端，那么使用管道可以显著减少网络延迟。

- 采用域名分区

  如果你的应用性能受限于默认的每来源 6 个连接，可以考虑将资源分散到多个来源。

- 打包资源以减少HTTP请求

  拼接和精灵图等技巧有助于降低协议开销，又能达成类似管道的性能提升。

- 嵌入小资源

  考虑直接在父文档中嵌入小资源，从而减少请求数量。

管道缺乏支持，而其他优化手段又各有各的利弊。事实上，这些优化措施如果过于激进或使用不当，反倒会伤害性能（这一点请参考第 11 章的深入讨论）。总之，要有务实的态度，通过度量来评估各种措施对性能的影响，在此基础上再迭代改进。天底下就没有包治百病的灵丹妙药。

对了，还有最后一招儿——升级到 HTTP 2.0。仅此一招儿抵得上前面提到的大多数针对 HTTP 1.x 的优化手段！ HTTP 2.0 不光能让应用加载更快，还能让开发更简单。

13.3 针对HTTP 2.0的优化建议

HTTP 2.0 的主要目标就是提升传输性能，实现客户端与服务器间较低的延迟和较高的吞吐量。显然，在 TCP 和 TLS 之上实现最佳性能，同时消除不必要的网络延迟，从来没有如此重要过。最低限度：

- 服务器的初始 cwnd 应该是 10 个分组；
- 服务器应该通过 ALPN（针对 SPDY 则为 NPN）协商支持 TLS；
- 服务器应该支持 TLS 恢复以最小化握手延迟。

简言之，请回顾 2.5 节“针对 TCP 的优化建议”和 4.7 节“针对 TLS 的优化建议”。要通过 HTTP 2.0 获得最佳性能，特别是从每个来源仅用一个连接的角度说，的确需要各层协议的紧密配合。

接下来，或许有点意外，那就是采用移动及其他经典的最佳做法：少发数据、削减请求，根据无线网络情况调整资源供给。不管使用什么版本的协议，减少传输的数据量和消除不必要的网络延迟，对任何应用都是最有效的优化手段。

最后，杜绝和忘记域名分区、文件拼接、图片精灵等不良的习惯，这些做法在 HTTP 2.0 之上完全没有必要。事实上，继续使用这些手段反而有害！可以利用 HTTP 2.0 内置的多路分发以及服务器推送等新功能。

致 HTTP 2.0 和 SPDY 早期采用者

官方 HTTP 2.0 标准还在制定中。在此期间，SPDY 是这个协议的“应用”版（参见 12.2 节的“HTTP 2.0 与 SPDY 共同进化”），它迅速得到各种客户端和服务器支持，并在几年里基于真实的应用提供反馈。如果你是一位早期采用者，那你肯定在一家好公司。

最后，虽然 SPDY 和 HTTP 2.0 规范并不完全同步，但它们却有相同的核心功能和优化。我们在这里以及之前几节讨论的内容对它们而言同样适用。

13.3.1 去掉对1.x的优化

针对 HTTP 2.0 和 HTTP 1.x 的优化策略没有什么重叠。因此，不仅不必担心 HTTP 1.x 协议的种种限制，而且要撤销原先那些必要的做法。

- 每个来源使用一个连接

  HTTP 2.0 通过将一个 TCP 连接的吞吐量最大化来提升性能。事实上，在 HTTP 2.0 之下再使用多个连接（比如域名分区）反倒成了一种反模式，因为多个连接会抵消新协议中首部压缩和请求优先级的效用。

- 去掉不必要的文件合并和图片拼接

  打包资源的缺点很多，比如缓存失效、占用内存、延缓执行，以及增加应用复杂性。有了 HTTP 2.0，很多小资源都可以并行发送，导致打包资源的效率反而更低。

- 利用服务器推送

  之前针对 HTTP 1.x 而嵌入的大多数资源，都可以而且应该通过服务器推送来交付。这样一来，客户端就可以分别缓存每个资源，并在页面间实现重用，而不必把它们放到每个页面里了。

要获得最佳性能，应该尽可能把所有资源都集中在一个域名之下。域名分区在 HTTP 2.0 之下属于反模式，对发挥协议的性能有害：分区是开始，之后影响会逐渐扩散。打包资源不会影响 HTTP 2.0 协议本身，但对缓存性能和执行速度有负面影响。

> 关于合并文件和拼接图片的负面影响，请参考 11.6 节“连接与拼合”，以及该节中的“计算图片对内存的需求”。

类似地，把嵌入资源改为服务器推送能提升客户端的缓存性能，又不会导致额外网络延迟（参见 12.3.7 节中的“实现 HTTP 2.0 服务器推送”）。事实上，由于 3G 和 4G 网络的往返时间更长，因而服务器推送对移动应用来说效果更明显。

HTTP 2.0 中的打包与协议开销

由于 HTTP 1.x 做不到多路复用，而且每次请求的协议开销很高，这才有了连接和拼合等打包技术。在 HTTP 2.0 之下，多路复用已经不成问题，首部压缩也可以降低每次 HTTP 请求要传输的元数据量，打包技术在多数情况下都不再需要了。

不过，请求开销只是减少了，并没有等于零。少数情况下，某些资源必须一块使用，而且更新也不频繁，此时使用打包技术仍然可以提升性能。但这些情况很少见，可以算作例外。具体措施可以通过性能度量确定。

13.3.2　双协议应用策略

遗憾的是，升级到 HTTP 2.0 不会在一夜之间完成。因此，很多应用都需要认真考虑双协议并存的部署策略，即同一个应用既能通过 HTTP 1.x 交付，也能通过 HTTP

2.0 交付，无需任何改动。然而，过于激进的 HTTP 1.x 优化可能伤害 HTTP 2.0 性能，反之亦然。

如果应用可以同时控制服务器和客户端，那倒简单了，因为它可以决定使用什么协议。但大多数应用不能也无法控制客户端，只有采用一种混合或自动策略，以适应两种协议并存的现实。下面我们就分析几种可能的情况。

- *相同的应用代码，双协议部署*

  相同的应用代码可能通过 HTTP 1.x 也可能通过 HTTP 2.0 交付。可能任何一种协议之下都达不到最佳性能，但可以追求性能足够好。所谓足够好，需要通过针对每一种应用单独度量来保证。这种情况下，第一步可以先撤销域名分区以实现 HTTP 2.0 交付。然后，随着更多用户迁移到 HTTP 2.0，可以继续撤销资源打包并尽可能利用服务器推送。

- *分离应用代码，双协议部署*

  根据协议不同分别交付不同版本的应用。这样会增加运维的复杂性，但实践中对很多应用倒是十分可行。比如，一台负责完成连接的边界服务器可以根据协商后的协议版本，把客户端请求引导至适当的服务器。

- *动态HTTP 1.x和HTTP 2.0优化*

  某些自动化的 Web 优化框架，以及开源及商业产品，都可以在响应请求时动态重写交付的应用代码（包括连接、拼合、分区，等等）。此时，服务器也可以考虑协商的协议版本，并动态采用适当的优化策略。

- *HTTP 2.0，单协议部署*

  如果应用可以控制服务器和客户端，那没理由不只使用 HTTP 2.0。事实上，如果真有这种可能，那就应该专一使用 HTTP 2.0。

选择路线时，要看当前的基础设施、应用的复杂程度，以及用户的构成。让人哭笑不得的是，那些在 HTTP 1.x 优化上投资很大的应用，反倒在这种情况下最难办。如果你能控制客户端，有自动的应用优化策略，或者没有使用任何特定于 1.x 的优化，那么就可以专注于 HTTP 2.0，而没有后顾之忧了。

使用 PageSpeed 实现动态优化

谷歌的 PageSpeed Optimization Libraries（PSOL）提供了 40 多种“Web 优化过滤器”的开源实现，可以集成到任何服务器运行时，动态应用各种优化策略。

在使用 PSOL 库的情况下，mod_pagespeed（Apache）和 ngx_pagespeed（Nginx）模块都可以基于指定的优化过滤器（如嵌入、压缩、拼接、分片等）实现动态重写，并优化资源交付方式。每次优化都在请求时动态应用（并被缓存），整个优化过程完全自动化了。

在动态优化下，服务器还可以根据所用协议，甚至用户代理的类型和版本调整优化策略。比如，可以配置 mod_pagespeed 模块，在客户端使用 HTTP 2.0 时跳过某些优化：

```
# 对 SPDY/HTTP 2.0 客户端禁用拼接
<ModPagespeedIf spdy>
  ModPagespeedDisableFilters combine_css,combine_javascript
</ModPagespeedIf>

# 只对 HTTP 1.x 客户端使用域名分区
<ModPagespeedIf !spdy>
  ModPagespeedShardDomain www.site.com s1.site.com,s2.site.com
</ModPagespeedIf>
```

使用 PageSpeed 这样的自动 Web 优化库，可以让我们省去不少麻烦，值得考虑。

13.3.3　1.x与2.0的相互转换

除了双协议优化策略，很多已部署的应用都需要在自己的应用服务器上采取一种折中方案：两端都是 HTTP 2.0 是追求最佳性能的目标，但（新增）一个转换层（图 13-2）也可以让 1.x 服务器利用 HTTP 2.0。

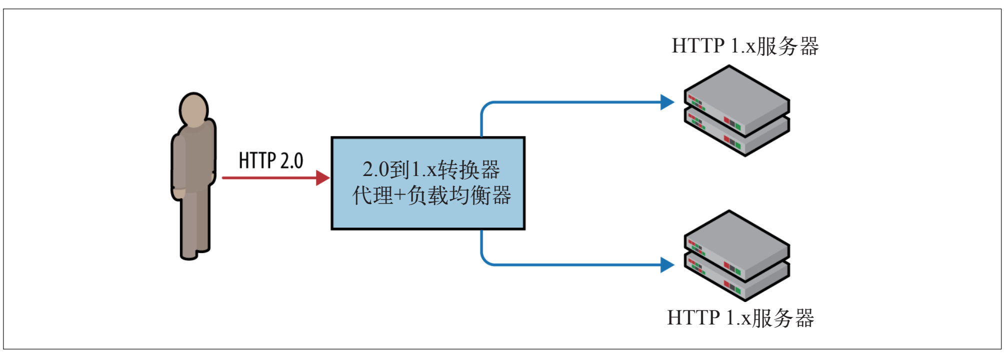


图 13-2：HTTP 2.0 到 1.x 的转换，即将流转换为 1.x 请求

一台居间服务器可以接受 HTTP 2.0 会话，处理之后再向既有基础设施分派 1.x 格式的请求。接到响应后，再将其转换成 HTTP 2.0 的流并返回客户端。通常，这是应用 HTTP 2.0 更新的最简单方式，因为这样可以重用已有的 1.x 基础设施，而且基本不用修改。

大多数支持 HTTP 2.0 的 Web 服务器默认都提供 2.0 到 1.x 的转换机制：2.0 会话终止于服务器（Apache 或 Nginx），如果服务器被配置为反向代理，那么分派给具体应用服务器的就是 1.x 请求。

然而，2.0 到 1.x 的这种简单策略并非长久之计。从很多方面来说，这种工作流实际是一种倒退。真正正确的做法，不是把优化的、可复用的会话转换成一系列 1.x 请求，因基础设施而废优化，而是相反：把接收到的 1.x 客户端请求转换成 2.0 流，并把我们的基础设施标准化，使其在任何时候都处理 2.0 会话。

为获得最佳性能，同时实现低延迟和实时的 Web 应用，应该要求我们的内部基础设施达到如下标准：

- 负载均衡器和代理与应用的连接应该持久化；
- 请求和响应流及多路复用应该是默认配置；
- 与应用服务器的通信应该基于消息；
- 客户端与应用服务器的通信应该是双向的。

端到端的 HTTP 2.0 会话符合上述所有条件，能实现对客户端以及数据中心内部的低延迟交付：无需定制的 RPC 层及相应机制，就能实现内部服务之间的通信，并获得理想的性能。简言之，不要把 2.0 降级到 1.x，这不是长久之计。长久之计是把 1.x 升级到 2.0，这样才能求得最佳性能。

13.3.4　评估服务器质量与性能

HTTP 2.0 服务器实现的质量对客户端性能影响很大。HTTP 服务器的配置当然是一个重要因素，但服务器实现逻辑的质量同样与优先级、服务器推送、多路复用等性能机制的发挥紧密相关。

- HTTP 2.0 服务器必须理解流优先级；
- HTTP 2.0 服务器必须根据优先级处理响应和交付资源；
- HTTP 2.0 服务器必须支持服务器推送；
- HTTP 2.0 服务器应该提供不同推送策略的实现。

HTTP 2.0 服务器的初级实现也能支持某些功能，但不能明确支持请求的优先级和服务器推送，可能导致次优性能。比如，发送大型、静态图片导致带宽饱和，而客户端又因为其他重要资源（如 CSS 或 JavaScript）被阻塞。

为尽可能获得最佳性能，HTTP 2.0 客户端必须是个“乐观主义者”：尽可能早地发送所有请求，然后完全听凭服务器的优化。事实上，HTTP 2.0 客户端对服务器的依赖程度较之以前更甚。

类似地，不同的服务器可能提供利用服务器推送的不同机制和策略（参见 12.3.7 节中的“实现 HTTP 2.0 服务器推送”）。如果说你的应用性能与 HTTP 2.0 服务器的质量紧密相关，是一点也不夸张的。

> 鉴于 HTTP 2.0 和 SPDY 的迅速发展，不同服务器（Apache、Nginx、Jetty 等）对 HTTP 2.0 的实现还处于不同的阶段。要了解它们当前支持的功能和最新消息，请查阅相应的文档和发版说明。

13.3.5　2.0与TLS

实践中，由于存在很多不兼容的中间代理，早期的 HTTP 2.0 部署必然依赖加密信道。这样一来，我们就面临两种可能出现 ALPN 协商和 TLS 终止的情况：

- TLS 连接可能会在 HTTP 2.0 服务器上终止；
- TLS 连接可能会在上游（如负载均衡器）上终止。

第一种情况要求 HTTP 2.0 服务器能够处理 TLS，除此之外就没有什么了。第二种情况复杂一些：TLS+ALPN 握手可能会在上游代理处终止（图 13-3），然后再从那里建立一条加密信道，或者直接将非加密的 HTTP 2.0 流发送到服务器。

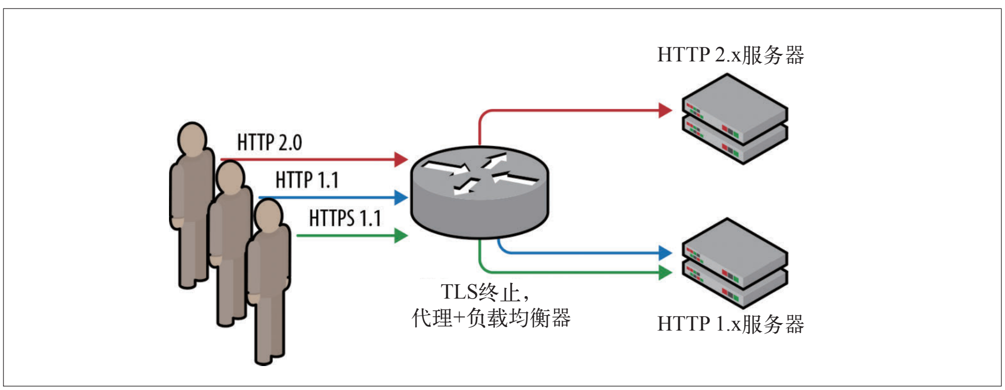


图 13-3：支持 TLS+ALPN 的负载均衡器

代理和应用服务器之间使用安全信道还是非加密信道，取决于应用：只要能控制中间设备，就可以保证未加密的帧不会被修改或丢弃。那么，虽然大多数 HTTP 2.0 服务器都应该支持 TLS+ALPN 协商，但它们同时也应该在不加密的情况下实现 HTTP 2.0 通信。

另外，智能负载均衡器也可以使用 TLS+ALPN 协商机制，根据协商后的协议，选择性地将不同的客户端路由到不同的服务器。

HAProxy 是一个流行的开源负载均衡器，同时支持 NPN 协商和基于协商后协议的路由。我的这篇文章是一个简单的介绍："Simple SPDY and NPN Negotiation with HAProxy"（http://hpbn.co/haproxy-npn）。

13.3.6　负载均衡器、代理及应用服务器

根据现有基础设施以及应用的复杂程度和规模，你的基础设施中可能需要一台或多台负载均衡器（图 13-4）或者 HTTP 2.0 代理。

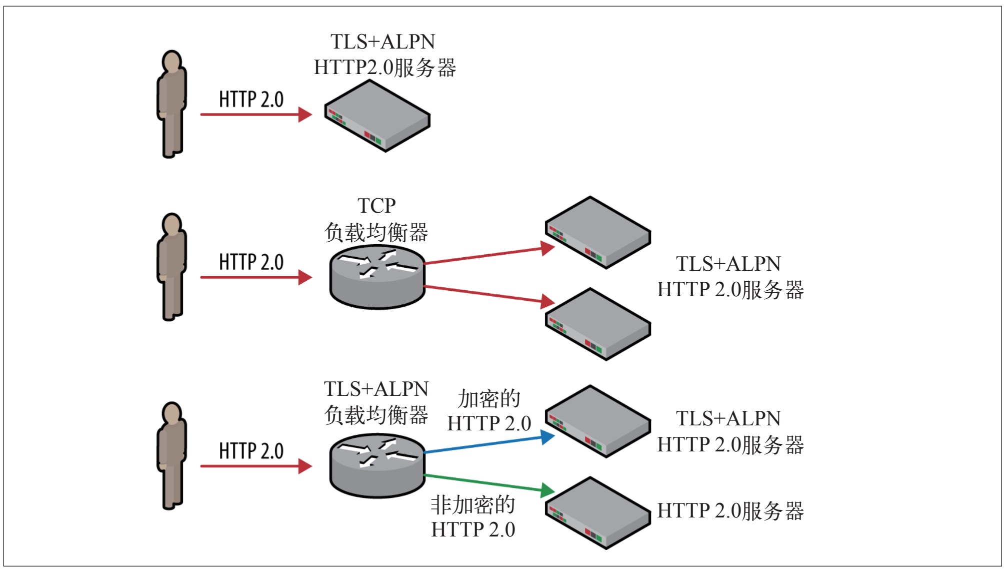


图 13-4：负载均衡器与 TLS 终止策略

最简单的情况下，HTTP 2.0 服务器与客户端直接对话，并负责完成 TLS 连接，进行 ALPN 协商，以及处理所有请求。

然而，一台服务器对于大型应用是不够的。大型应用必须要添加一台负载均衡器，以分流大量请求。此时，负载均衡器可以终止 TLS 连接（参见 13.3.5 节"2.0 与 TLS"），也可以经过配置作为 TCP 代理并直接将加密数据发送给应用服务器。

很多云提供商也会提供负载均衡器服务。然而，这些负载均衡器大多支持 TLS 终止，却不支持 ALPN 协商，而这对于通过 TLS 实现 HTTP 2.0 通信是必需的。在这种情况下，应该将负载均衡器配置为 TCP 代理，即通过它们将加密数据发送给应用服务器，让应用服务器完成 TLS+ALPN 协商。

实践中，要回答的最重要的一个问题，就是你的基础设施中的哪个组件负责终止TLS 连接，以及它是否能够执行必要的 ALPN 协商？

- 要在 TLS 之上实现 HTTP 2.0 通信，终端服务器必须支持 ALPN；
- 尽可能在接近用户的地方终止 TLS，参见 4.7.2 节“尽早完成（握手）”；
- 如果无法支持 ALPN，那么选择 TCP 负载均衡模式；
- 如果无法支持 ALPN 且 TCP 负载均衡也做不到，那么就退而求其次，在非加密信道上使用 HTTP 的 `Upgrade` 流，参见 12.3.9 节“有效的 HTTP 2.0 升级与发现”。

第四部分

浏览器API与协议

第 14 章

浏览器网络概述

作为一个平台，现代浏览器是专门设计用来快速、高效、安全地交付 Web 应用的。事实上，在其表面之下，现代浏览器完全是一个囊括数百个组件的操作系统，包括进程管理、安全沙箱、分层的优化缓存、JavaScript 虚拟机、图形渲染和 GPU 管道、存储系统、传感器、音频与视频、网络机制，等等。

显然，浏览器乃至运行在其中的应用的性能，取决于若干组件：解析、布局、HTML 与 CSS 的样式计算、JavaScript 执行速度、渲染管道，当然还有网络相关各层协议的配合。其中每个组件的角色都很重要，而网络组件通常是加倍重要，因为浏览器慢就慢在等待网络资源上，等待造成后续环节被阻塞！

自然地，现代浏览器对各层网络协议的实现也就远不止一个套接字管理器那么简单。从外界看，可以把网络组件当成一个简单的获取机制，但从内部看，它本身又是一个平台的概念（图 14-1），有自己的优化条件、API 和服务。

设计 Web 应用的时候，我们不必关心个别的 TCP 或 UDP 套接字，浏览器会替我们管理它们。而且，网络组件会帮我们施加恰当的连接限制、格式化请求、隔离应用、管理代理、缓存，等等。在隐藏了这些复杂性的基础上，我们才可以专注于自己的应用逻辑。

可是，眼不见不意味着心不烦！前面已经讨论过了，理解 TCP、HTTP，还有移动网络的性能特点，有助于我们构建更快的应用。同理，理解如何最恰当地利用浏览器的网络 API、协议和服务，照样能给应用带来显著的性能提升。

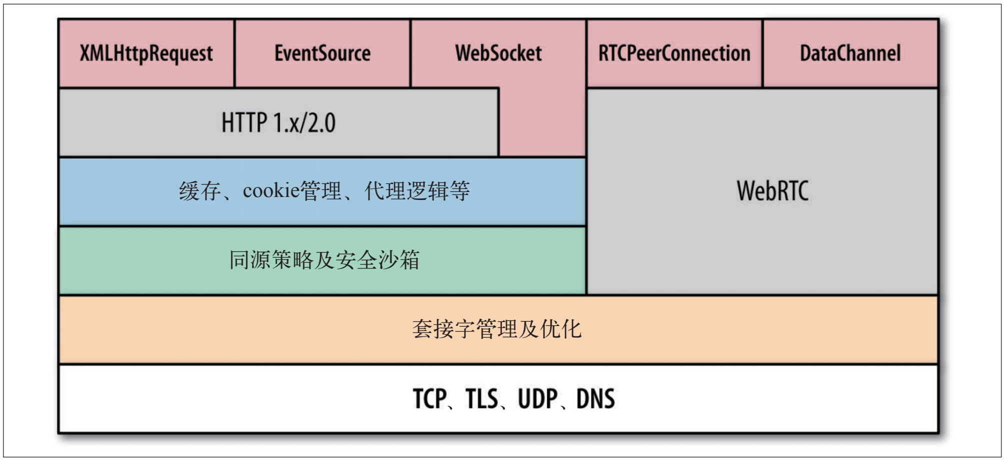


图 14-1：高层浏览器网络 API、协议和服务

14.1　连接管理与优化

运行在浏览器中的 Web 应用并不负责管理个别网络套接字的生命周期，这是好事。通过把这个任务委托给浏览器，可以自动化很多重要的性能优化任务，包括套接字重用、请求优先级排定、晚绑定、协议协商、施加连接数限制，等等。事实上，浏览器是有意把请求管理生命周期与套接字管理分开的。这一点很微妙，但却至关重要。

套接字是以池的形式进行管理的（图 14-2），即按照来源，每个池都有自己的连接限制和安全约束。挂起的请求是排好队的、有优先次序的，然后再适时把它们绑定到池中个别的套接字上。除非服务器有意关闭连接，否则同一个套接字可以自动用于多个请求！

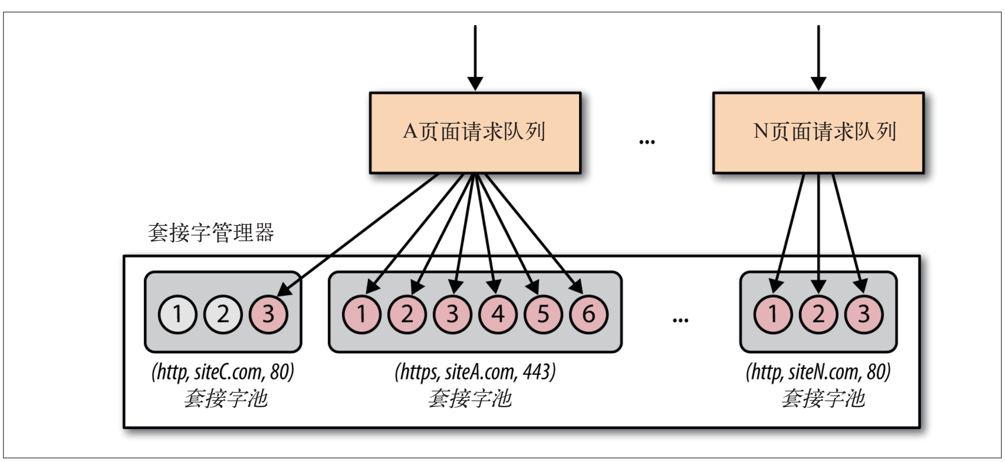


图 14-2：自动管理的套接字池在所有浏览器进程间共享

- 来源

  由应用协议、域名和端口三个要件构成，比如 (http, www.example.com, 80) 与 (https, www.example.com, 443) 就是两个不同的来源。

- 套接字池

  属于同一个来源的一组套接字。实践中，所有主流浏览器的最大池规模都是 6 个套接字。

自动化的套接字池管理会自动重用 TCP 连接，从而有效保障性能（参见 11.1 节“持久连接的优点”）。除此之外，这种架构设计还提供了其他优化的机会：

- 浏览器可以按照优先次序发送排队的请求；
- 浏览器可以重用套接字以最小化延迟并提升吞吐量；
- 浏览器可以预测请求提前打开套接字；
- 浏览器可以优化何时关闭空闲套接字；
- 浏览器可以优化分配给所有套接字的带宽。

简单来说，浏览器的网络组件是我们交付高性能应用的同盟。刚刚介绍的这些功能，没有一项需要我们参与！但这并不是说我们不能再帮助浏览器。我们的设计决策对应用性能同样起着至关重要的作用，因为这些决策会影响网络通信的模式、类型和传输频率，以及协议选择和服务器端的性能优化。

谷歌 Chrome 的推测性网络优化

我们已经知道了，现代浏览器的网络组件并非一个套接字管理器那么简单。但是，即使如此有时候也足以客观地评价现代浏览器中的某些优化技术。

比如，你使用谷歌 Chrome 浏览器的次数越多，它的速度就会越快。Chrome 会学习访问过的站点的拓扑，以及常见的浏览模式，然后利用这些信息进行各种“推测性优化”，以预测用户下一步的操作，从而消除不必要的网络延迟：DNS 预解析、TCP 预连接、页面预渲染，等等。像鼠标悬停在链接上这么个简单的动作，就可以触发浏览器向其网络组件的“预测器”发送信号，后者则会依据过往的性能数据选择最佳的优化措施。

要了解这方面更多信息，请参考 10.5 节“针对浏览器的优化建议”。如果你对 Chrome 浏览器的网络优化技术感兴趣，可以看看这篇文章“High Performance Networking in Google Chrome”：http://hpbn.co/chrome-networking。

14.2 网络安全与沙箱

将个别套接字的管理任务委托给浏览器还有另一个重要的用意：可以让浏览器运用沙箱机制，对不受信任的应用代码采取一致的安全与策略限制。比如，浏览器不允许直接访问原始网络套接字 API，因为这样给恶意应用向任意主机发起任意请求（端口扫描、连接邮件服务器或发送未知消息）提供可乘之机。

- 连接限制

  浏览器管理所有打开的套接字池并强制施加连接数限制，保护客户端和服务器的资源不会被耗尽。

- 请求格式化与响应处理

  浏览器格式化所有外发请求以保证格式一致和符合协议的语义，从而保护服务器。类似地，响应解码也会自动完成，以保护用户。

- TLS协商

  浏览器执行 TLS 握手和必要的证书检查。任何证书有问题（比如服务器正在使用自己签发的证书），用户都会收到通知。

- 同源策略

  浏览器会限制应用只能向哪个来源发送请求。

以上列出的安全限制机制只是一部分，但已经可以体现“最低特权”（least privilege）原则了。浏览器只向应用代码公开那些必要的 API 和资源：应用提供数据和 URL，浏览器执行请求并负责管理每个连接的整个生命周期。

> 有必要提一句，并没有单独一条原则叫“同源策略”。实际上，这是一组相关的机制，涉及对 DOM 访问、cookie 和会话状态管理、网络及其他浏览器组件的限制。
>
> 要全面解析浏览器安全，可能需要另外一本书。如果你真想了解这方面信息，可以看看 Michal Zalewski 的 *The Tangled Web: A Guide to Securing Modern Web Applications*[1]。

14.3 资源与客户端状态缓存

最好最快的请求是没有请求。在分派请求之前，浏览器会自动检查其资源缓存，执行必要的验证，然后在满足限制条件的情况下返回资源的本地副本。类似地，如果

注 1：中文版《Web 之困：现代 Web 应用安全指南》由机械工业出版社出版。——译者注

某本地资源不在缓存中，那么浏览器就会发送网络请求，将响应自动填充到缓存中，以备后续访问使用。

- 浏览器针对每个资源自动执行缓存指令。
- 浏览器会尽可能恢复失效资源的有效性。
- 浏览器会自动管理缓存大小及资源回收。

高效、最优地管理缓存很困难。所幸，浏览器会替我们照管这一切，我们要做的，只是确保服务器返回适当的缓存指令（参见 13.1.1 节“在客户端缓存资源”）。你已经让服务器对页面的所有资源都返回 `Cache-Control`、`ETag` 和 `Last-Modified` 等响应首部了，对不？

最后，浏览器还有一个经常被人忽视的重要功能，那就是提供会话认证和 cookie 管理。浏览器为每个来源维护着独立的 cookie 容器，为读写新 cookie、会话和认证数据提供必要的应用及服务器 API，还会为我们自动追加和处理 HTTP 首部，让一切都自动化。

> 举一个简单但直观的例子，它能说明把会话状态管理委托给浏览器的好处：认证的会话可以在多个标签页或浏览器口间共享，反之亦然；如果用户在某个标签页中退出，那么其他所有打开窗口中的会话都将失效。

14.4 应用API与协议

在浏览器提供的网络服务的最上层，就是应用 API 和协议。前面介绍了下层提供的各种重要服务：套接字和连接管理、请求和响应处理、各种安全机制、缓存，等等。我们每次发起 HTTP 或 XMLHttpRequest 请求，或者长 Server-Sent Event 或 WebSocket 会话，或者打开 WebRTC 连接，都需要与其中一些或全部底层服务打交道。

不存在哪个协议或 API 最好的问题。每个稍微复杂点的应用都会基于不同的需求用到各种传输机制，包括读写浏览器缓存、协议开销、消息延迟、可靠性、数据传输类型，等等。某些协议的交付延迟可能短一些（比如 Server-Sent Events、WebSocket），但却不能满足其他条件，比如利用浏览器缓存，或者在所有场景下支持高效的二进制传输（表 14-1）。

表14-1：XHR、SSE和WebSocket的高级特性

	XMLHttpRequest	Server-Sent Event	WebSocket
请求流	否	否	是
响应流	受限	是	是
分帧机制	HTTP	事件流	二进制分帧
二进制数据传输	是	否（base64）	是
压缩	是	是	受限
应用传输协议	HTTP	HTTP	WebSocket
网络传输协议	TCP	TCP	TCP

我们在这个表中有意忽略了 WebRTC，因为那是一种端到端的交付模型，与 XHR、SSE 和 WebSocket 协议有着根本的不同。

这里比较的高级特性并不完整（其余内容将在接下来几章讨论），但也足以表明每种协议之间的大量差异。理解了每种协议的长处和短处，根据应用的需求恰当运用它们，就可以摆脱贫乏的用户体验，打造出高性能应用。

第15章

XMLHttpRequest

`XMLHttpRequest`（XHR）是浏览器层面的 API，可以让开发人员通过 JavaScript 实现数据传输。XHR 是在 Internet Explorer 5 中首次亮相的，后来成为 AJAX（Asynchronous JavaScript and XML）革命的核心技术，是今天几乎所有 Web 应用必不可少的基本构件。

> XMLHTTP 改变了一切。它让 DHTML 中的 D 变得名副其实。它让我们能异步从服务器获取数据，同时在客户端保持文档状态……Outlook Web Access（OWA）团队希望在浏览器中构建一个类似 Win32 应用的想法使其得以进入 IE，而这才有了后来的 AJAX。
>
> ——Jim Van Eaton
>
> *Outlook Web Access: A catalyst for web evolution*

XHR 诞生前，网页要获取客户端和服务器的任何状态更新，都必须刷新一次。有了 XHR，这个过程就可以异步实现，而且完全通过应用的 JavaScript 代码完成。XHR 是让我们从制作网页转换为开发交互应用的根本技术。

然而，XHR 的能力不仅仅表现在能实现浏览器的异步通信，还表现在它极大地简化了这个异步通信过程。XHR 是浏览器提供的应用 API，这就意味着浏览器会自动帮我们完成所有底层的连接管理、协议协商、HTTP 请求格式化，以及更多工作：

- 浏览器管理着连接建立、套接字池和连接终止；
- 浏览器决定最佳的 HTTP（S）传输协议（HTTP 1.0、1.x 和 2.0）；
- 浏览器处理 HTTP 缓存、重定向和内容类型协商；

- 浏览器保障安全、验证和隐私；
- 浏览器……

不用关心这些底层细节，那么我们就可以把时间和精力放在应用的业务逻辑上，只要发起请求、管理进度，然后处理服务器返回的数据即可。简单的 API 加上所有浏览器的支持，使得 XHR 成为了浏览器网络开发中的“瑞士军刀”。

结果，几乎所有网络应用（脚本下载、上传、流传输，甚至实时通知），都能够或者已经都通过 XHR 实现。当然，这并不是说 XHR 在任何场景中都是最有效的传输方式（事实上，后面我们会讨论，多数情况下并非如此），但无论如何它都经常被作为旧版客户端的后备传输方式，这些旧客户端通常没有实现较新的浏览器网络 API。知道了这一点，下面就来了解一下 XHR 最新的功能，它的适用场景，以及我们在性能优化方面能做什么和不能做什么。

> 详尽讨论 XHR API 超出了我们的范畴，我们只讨论性能相关的话题！请参考 W3C 官方对 XMLHttpRequest API 的描述：http://www.w3.org/TR/XMLHttpRequest/。

15.1 XHR简史

尽管名字里有 XML 的 X，XHR 也不是专门针对 XML 开发的。这只是因为 Internet Explorer 5 当初发布它的时候，把它放到 MSXML 库里，这才“继承”了这个 X：

> 那是值得纪念的日子，就在发版前几天，又增加了一些关键的功能……我知道 MSXML 库要随 IE 一起发布，我跟 XML 团队关系还不错，觉得他们或许能帮个忙。我找到当时的团队领导 Jean Paoli，很快就说服他同意把它作为 MSXML 库的一部分发布。它其实主要跟 HTTP 相关，跟 XML 基本上没什么关系。仅仅是为了让它能早点发布，我才硬给它起了一个带XML的名字。
>
> ——Alex Hopmann
> *The story of XMLHTTP*

Mozilla 按照微软的实现也实现了自己的 XHR，并将其命名为 XMLHttpRequest。Safari、Opera 和其他浏览器也紧随其后，于是 XHR 成为了所有主流浏览器中的事实标准。W3C 针对 XHR 的官方工作草案发布于 2006 年，而这已经是 XHR 得到广泛应用以后的事了！

虽然它在 AJAX 革命中扮演了至关重要的角色，但 XHR 的早期版本确实能力有限：只能传输文本，处理上传的能力不足，而且不能处理跨域请求。为解决这些问题，W3C 于 2008 年发布了“XMLHttpRequest Level 2”草案，新增了如下一些新功能：

- 支持请求超时；
- 支持传输二进制和文本数据；
- 支持应用重写媒体类型和编码响应；
- 支持监控每个请求的进度事件；
- 支持有效的文件上传；
- 支持安全的跨来源请求。

2011 年，“XMLHttpRequest Level 2”规范与原来的 XMLHttpRequest 工作草案合并。此后，无论人们提及 XHR 时说 Level 1 还是 Level 2，其实已经没有关系了。今天，只有一个统一的 XHR 规范。而所有新的 XHR2 功能，都是通过同一个 `XMLHttpRequest` API 提供的：接口不变，功能增强。

> 新 XHR2 功能目前已经得到所有现代浏览器支持，参见：caniuse.com/xhr2。此后，只要我们提到 XHR，指的都是 XHR2 标准。

15.2 跨源资源共享（CORS）

XHR 是一个浏览器层面的 API，向我们隐藏了大量底层处理，包括缓存、重定向、内容协商、认证，等等。这样做有两个目的。第一，XHR 的 API 因此非常简单，开发人员可以专注业务逻辑。其次，浏览器可以采用沙箱机制，对应用代码强制施加一套安全限制。

XHR 接口强制要求每个请求都严格具备 HTTP 语义：应用提供数据和 URL，浏览器格式化请求并管理每个连接的完整生命周期。类似地，虽然 XHR API 允许应用添加自定义的 HTTP 首部（通过 `setRequestHeader()` 方法），同时也有一些首部是应用代码不能设定的：

- Accept-Charset、Accept-Encoding、Access-Control-*
- Host、Upgrade、Connection、Referer、Origin
- Cookie、Sec-*、Proxy-* 以及很多其他首部

浏览器会拒绝对不安全首部的重写，以此保证应用不能假扮用户代理、用户或请求来源。事实上，保护来源（Origin）首部特别重要，因为这是对所有 XHR 请求应用“同源策略”的关键。

> 一个“源”由应用协议、域名和端口这三个要件共同定义。比如，(http, example.com, 80) 和 (https, example.com, 443) 就是不同的源。更多信息，请参考“The Web Origin Concept”（http://tools.ietf.org/html/draft-abarth-origin）。

同源策略的出发点很简单：浏览器存储着用户数据，比如认证令牌、cookie 及其他私有元数据，这些数据不能泄露给其他应用。如果没有同源沙箱，那么 example.com 中的脚本就可以访问并操纵 thirdparty.com 的用户数据！

为解决这个问题，XHR 的早期版本都限制应用只能执行同源请求，即新请求的来源必须与旧请求的来源一致：来自 example.com 的 XHR 请求，只能从 example.com 请求其他资源。如果后续请求不同源，浏览器就拒绝该 XHR 请求并报错。

可是，在某些必要的情况下，同源策略也会给更好地利用 XHR 带来麻烦：如果服务器想要给另一个网站中的脚本提供资源怎么办？这就是 Cross-Origin Resource Sharing（跨源资源共享，CORS）的来由！ CORS 针对客户端的跨源请求提供了安全的选择同意机制：

```
// 脚本来源：(http, example.com, 80)
var xhr = new XMLHttpRequest();
xhr.open('GET', '/resource.js'); ❶
xhr.onload = function() { ... };
xhr.send();

var cors_xhr = new XMLHttpRequest();
cors_xhr.open('GET', 'http://thirdparty.com/resource.js'); ❷
cors_xhr.onload = function() { ... };
cors_xhr.send();
```

❶ 同源 XHR 请求
❷ 跨源 XHR 请求

CORS 请求也使用相同的 XHR API，区别仅在于请求资源用的 URL 与当前脚本并不同源。在前面的例子中，当前执行的脚本来自 (http, example.com, 80)，而第二个 XHR 请求访问的 resource.js 则来自 (http, thirdparty.com, 80)。

针对 CORS 请求的选择同意认证机制由底层处理：请求发出后，浏览器自动追加受保护的 Origin HTTP 首部，包含着发出请求的来源。相应地，远程服务器可以检查 Origin 首部，决定是否接受该请求，如果接受就返回 Access-Control-Allow-Origin 响应首部：

```
=> 请求
GET /resource.js HTTP/1.1
Host: thirdparty.com
Origin: http://example.com ❶
...

<= 响应
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://example.com ❷
...
```

❶ Origin 首部由浏览器自动设置

❷ 选择同意首部由服务器设置

在前面的例子中，thirdparty.com 决定同意与 example.com 跨源共享资源，因此就在响应中返回了适当的访问控制首部。假如它选择不同意接受这个请求，那么只要不在响应中包含 *Access-Control-Allow-Origin* 首部即可。这样，客户端的浏览器就会自动将发出的请求作废。

> 如果第三方服务器不支持 CORS，那么客户端请求同样会作废，因为客户端会验证响应中是否包含选择同意的首部。作为一个特例，CORS 还允许服务器返回一个通配值（`Access-Control-Allow-Origin: *`），表示它允许来自任何源的请求。不过，在启用这个选项前，请大家务必三思！

这就是全部了吧？准确地讲，不是。因为 CORS 还会提前采取一系列安全措施，以确保服务器支持 CORS：

- CORS 请求会省略 cookie 和 HTTP 认证等用户凭据；
- 客户端被限制只能发送“简单的跨源请求”，包括只能使用特定的方法（GET、POST 和 HEAD），以及只能访问可以通过 XHR 发送并读取的 HTTP 首部。

要启用 cookie 和 HTTP 认证，客户端必须在发送请求时通过 XHR 对象发送额外的属性（`withCredentials`），而服务器也必须以适当的首部（*Access-Control-Allow-Credentials*）响应，表示它允许应用发送用户的隐私数据。类似地，如果客户端需要写或者读自定义的 HTTP 首部，或者想要使用“不简单的方法”发送请求，那么它必须首先要获得第三方服务器的许可，即向第三方服务器发送一个预备（preflight）请求：

```
=> 预备请求
OPTIONS /resource.js HTTP/1.1 ❶
Host: thirdparty.com
Origin: http://example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: My-Custom-Header
...

<= 预备响应
HTTP/1.1 200 OK ❷
Access-Control-Allow-Origin: http://example.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: My-Custom-Header
...

（正式的 HTTP 请求）❸
```

❶ 验证许可的预备 `OPTIONS` 请求
❷ 第三方源的成功预备响应
❸ 实际的 CORS 请求

W3C 官方的 CORS 规范规定了何时何地必须使用预备请求："简单的"请求可以跳过它，但很多条件下这个请求都是必需的，因此也会为验证许可而增加仅有一次往返的网络延迟。好在，只要完成预备请求，客户端就会将结果缓存起来，后续请求就不必重复验证了。

> CORS 得到了所有现代浏览器支持，参见：caniuse.com/cors。要全面了解 CORS 的各种策略及实现，请参考 W3C 官方标准（http://www.w3.org/TR/cors/）。

15.3 通过XHR下载数据

XHR 既可以传输文本数据，也可以传输二进制数据。事实上，浏览器可以自动为各种原生数据类型提供编码和解码服务，因此应用在直接将这些数据传给 XHR 时就已经编码 / 解码好了，反之亦然。浏览器可以自动解码的数据类型如下。

- `ArrayBuffer`
  固定长度的二进制数据缓冲区。

- `Blob`
  二进制大对象或不可变数据。

- `Document`
  解析后得到的 HTML 或 XML 文档。

- `JSON`
  表示简单数据结构的 JavaScript 对象。

- `Text`
  简单的文本字符串。

浏览器可以依靠 HTTP 的 `content-type` 首部来推断适当的数据类型（比如把 `application/json` 响应解析为 JSON 对象），应用也可以在发起 XHR 请求时显式重写数据类型：

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/images/photo.webp');
```

```
xhr.responseType = 'blob'; ❶

xhr.onload = function() {
  if (this.status == 200) {
    var img = document.createElement('img');
    img.src = window.URL.createObjectURL(this.response); ❷
    img.onload = function() {
        window.URL.revokeObjectURL(this.src); ❸
    }
    document.body.appendChild(img);
  }
};

xhr.send();
```

❶ 将返回数据类型设置为 Blob

❷ 基于返回的对象创建唯一的对象 URI 并设置为图片的源

❸ 图片加载完毕后立即释放对象

注意，这里我们在以原生格式传输一张图片，没有使用 base64 编码，也没有使用数据 URI，而是在页面中添加了一个 <img> 元素。这样在 JavaScript 中处理接收到的二进制数据不会产生任何网络传输开销和编码开销！ XHR API 让我们得以通过脚本高效、动态地开发应用，无论操作什么数据类型都没问题，全部用 JavaScript 搞定！

> 这里的二进制大对象接口（Blob）属于 HTML5 的 File API，就像一个不透明的引用，可以指向任何数据块（二进制或文本）。这个对象本身没有太多功能，只能查询其大小、MIME 类型，或将它切分成更小的块。这个对象存在的真正目的，是作为各种 JavaScript API 之间的一种高效的互操作机制。

15.4 通过XHR上传数据

通过 XHR 上传任何类型的数据都很简单，而且高效。事实上，上传不同类型数据的代码都一样，只不过最后在调用 XHR 请求对象的 send() 方法时，要传入相应的数据对象。剩下的事就都由浏览器处理了：

```
var xhr = new XMLHttpRequest();
xhr.open('POST','/upload');
xhr.onload = function() { ... };
xhr.send("text string"); ❶

var formData = new FormData(); ❷
```

```
formData.append('id', 123456);
formData.append('topic', 'performance');

var xhr = new XMLHttpRequest();
xhr.open('POST', '/upload');
xhr.onload = function() { ... };
xhr.send(formData); ❸

var xhr = new XMLHttpRequest();
xhr.open('POST', '/upload');
xhr.onload = function() { ... };
var uInt8Array = new Uint8Array([1, 2, 3]); ❹
xhr.send(uInt8Array.buffer); ❺
```

❶ 把简单的文本字符串上传到服务器
❷ 通过 FormData API 动态创建表单数据
❸ 向服务器上传 `multipart/form-data` 对象
❹ 创建无符号、8 字节整型的有类型数组（ArrayBuffer）
❺ 向服务器上传字节块

XHR 对象的 `send()` 方法可以接受 `DOMString`、`Document`、`FormData`、`Blob`、`File` 及 `ArrayBuffer` 对象，并自动完成相应的编码，设置适当的 HTTP 内容类型 (content-type)，然后再分派请求。需要发送二进制 `Blob` 或上传用户提交的文件？简单，取得对该对象的引用，传给 XHR。事实上，多写几行代码，还可以把大文件切成几小块：

```
var blob = ...; ❶

const BYTES_PER_CHUNK = 1024 * 1024; ❷
const SIZE = blob.size;

var start = 0;
var end = BYTES_PER_CHUNK;

while(start < SIZE) { ❸
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/upload');
  xhr.onload = function() { ... };

  xhr.setRequestHeader('Content-Range', start+'-'+end+'/'+SIZE); ❹
  xhr.send(blob.slice(start, end)); ❺

  start = end;
  end = start + BYTES_PER_CHUNK;
}
```

❶ 任意数据（二进制或文本）的二进制对象
❷ 将块大小设置为 1 MB

❸ 以 1 MB 为步长迭代数据块
❹ 告诉服务器上传的数据范围（开始位置 – 结束位置 / 总大小）
❺ 通过 XHR 上传 1 MB 大小的数据片段

XHR 不支持请求流，这意味着在调用 `send()` 时必须提供完整的文件。不过，前面的例子示范了一个简单的解决方案：切分文件，然后通过多个 XHR 请求分段上传。这种实现方案当然不能替代真正的请求流 API，但对某些应用来说却是一个可行的方案。

> 切分大文件上传是个不错的技巧，适合连接不稳定或经常中断的场景。此时，假如某个块由于掉线而上传失败，应用可以随后只重新上传该块，而不必重新上传整个大文件。

15.5　监控下载和上传进度

网络连接可能会间歇性中断，而延迟和带宽也高度不稳定。因此，我们怎么知道 XHR 请求成功了，超时了，还是失败了？ XHR 对象提供了一个方便的 API，用于监控进度事件（表 15-1），这些事件代表请求的当前状态。

表15-1：XHR的进度相关事件

事件类型	说明	触发次数
`loadstart`	传输已开始	一次
`progress`	正在传输	零或多次
`error`	传输出错	零或多次
`abort`	传输终止	零或多次
`load`	传输成功	零或多次
`loadend`	传输完成	一次

每个 XHR 请求开始时都会触发 `loadstart` 事件，而结束时都会触发 `loadend` 事件。在这两事件之间，还可能触发一或多个其他事件，表示传输状态。因此，要监控进度，可以在 XHR 对象上注册一系列 JavaScript 事件监听器：

```
var xhr = new XMLHttpRequest();
xhr.open('GET','/resource');
xhr.timeout = 5000; ❶

xhr.addEventListener('load', function() { ... }); ❷
xhr.addEventListener('error', function() { ... }); ❸

var onProgressHandler = function(event) {
```

```
    if(event.lengthComputable) {
      var progress = (event.loaded / event.total) * 100; ❹
      ...
    }
  }

  xhr.upload.addEventListener('progress', onProgressHandler); ❺
  xhr.addEventListener('progress', onProgressHandler); ❻
  xhr.send();
```

❶ 设置请求的超时时间为 5000 ms（默认无超时限制）
❷ 为请求成功注册回调
❸ 为请求失败注册回调
❹ 计算传输进度
❺ 为上传进度事件注册回调
❻ 为下载进度事件注册回调

无论 `load` 和 `error` 中的哪一个被触发了，都代表 XHR 传输的最终状态，而 `progress` 事件则可能触发任意多次，这就为监控传输状态提供了便利：我们可以比较 `loaded` 与 `total` 属性，估算传输完成的数据比例。

> 要估算传输完成的数据量，服务器必须在其响应中提供内容长度（Content-Length）首部。而对于分块数据，由于响应的总长度未知，因此就无法估计进度了。
>
> 另外，XHR 请求默认没有超时限制，这意味着一个请求的“进度”可以无限长。作为最佳实践，一定要为应用设置合理的超时时间，并适当处理错误。

15.6　通过XHR实现流式数据传输

在某些场景下，应用可能需要或者应该递增地流式处理数据。比如，等到客户端数据可用时上传，或者一边从服务器下载一边处理数据。可惜的是，尽管这种场景很重要，但时至今日仍然没有一种简单有效和跨浏览器的 API 来实现 XHR 流：

- 上传时，`send` 方法只接受完整的载荷；
- `response`、`responseText` 和 `responseXML` 属性也不是为流设计的。

在 XHR 规范中，流式数据处理从未成为官方正式考虑的使用场景。于是，除了手工把要上传的数据切分，再利用多个 XHR 请求迭代上传之外，没有其他 API 可以实现客户端与服务器间的流式数据传输。类似地，虽然 XHR2 规范提供了读取服务器部分响应的能力，但实现的效率却很低，而且有诸多限制。这一点确实令人沮丧。

不过，也不要彻底不抱希望。没有把流作为 XHR 的第一类场景考虑是一个公认的缺漏，因此目前正有人在积极地解决这个问题：

> Web 应用必须有能力获得并操作各种形式的数据，包括随着时间推移逐渐可用的一系列数据。本规范定义了流的基本表示法、流触发的错误，以及通过编程方式读取和创建流的方式。
>
> ——W3C Streams API

XHR 加上 Streams API 可以让浏览器支持高效的 XHR 流。可是，Streams API 目前还处于研讨阶段，没有一个浏览器支持它。因此，我们暂时就没有别的办法了，对吧？嗯，也不完全对。如前所述，通过 XHR 实现流式上传还不行，通过 XHR 实现流式下载却得到了浏览器有限的支持：

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/stream');
xhr.seenBytes = 0;

xhr.onreadystatechange = function() { ❶
  if(xhr.readyState > 2) {
    var newData = xhr.responseText.substr(xhr.seenBytes); ❷
    // 处理 newData

    xhr.seenBytes = xhr.responseText.length; ❸
  }
};

xhr.send();
```

❶ 预订状态和进度通知
❷ 从部分响应中提取新数据
❸ 更新处理的字节偏移量

这个例子在多数现代浏览器中都可以运行，但性能并不理想。另外，还有很多关于实现的问题和注意事项。

- 我们是手工跟踪看到的字节的偏移量，然后再手工切分数据：`responseText` 则缓冲了完整的响应！对于少量数据传输而言，这不是问题。但如果下载的数据很大，特别是在内存十分有限的设备比如手机上，这就是问题了。释放缓冲数据的唯一方式就是完成当前请求，并且打开一个新请求。
- 部分响应只能通过读取 `responseText` 属性获取，因此也就只能局限于文本数据了。没有办法部分读取二进制数据的响应。

- 读取完部分数据后，我们必须自己标识数据的界限：应用代码必须定义自己的数据格式，然后再缓冲并解析数据流，提取出相应的信息。
- 浏览器缓冲收到数据的方式也不相同：有的会立即释放，而有的只缓冲小响应，对较大的响应块则立即释放。
- 浏览器允许递增读取的数据类型也不一样：有的允许 `text/html`，而有的只允许 `application/x-javascript`。

简言之，目前基于 XHR 的流实现起来既麻烦，效率又低。更糟糕的是，由于缺乏规范，浏览器实现一家一个样。结论就是，在 Streams API 规范正式推出之前，XHR 并不适合用来实现流式数据处理。

> 没有必要失望！虽然 XHR 满足不了我们的要求，我们还有其他办法，而且是专门为流式数据处理设计的：Server-Sent Events 提供方便的流 API，用于从服务器向客户端发送文本数据，而 WebSocket 则提供了高效、双向的流机制，而且同时支持二进制和文本数据。

专有API和对XHR流的扩展

Firefox 和 Internet Explorer 都提供了定制的“XHR 流扩展”：

- Firefox 支持 `moz-chunked-text` 和 `moz-chunked-arraybuffer`；
- Internet Explorer 支持 `ms-stream`。

通过将 XHR 对象上的 `responseType` 属性设置为前述数据类型，这两个浏览器就可以不缓冲整个响应，同时允许通过 XHR 对象递增地读取二进制响应。遗憾的是，Chrome、Opera 及其他主流浏览器还没有类似的 API。因此，XHR 流对于跨浏览器应用来说，仍然不可行。

15.7　实时通知与交付

XHR 提供了一种简单有效的客户端与服务器同步的方式：必要时，客户端可以向服务器发送一个 XHR 请求，以更新服务器上的相应数据。然而，实现同样但相反的操作却要困难一些。如果服务器的数据更新了，那怎么通知客户端呢？

HTTP 没有提供服务器向客户端发起连接的方式。因此，为实时接收通知，客户端要么必须轮询服务器，要么就得利用流式传输让服务器推送通知。如前所述，主流浏览器对 XHR 流的支持有限，那我们就只能使用 XHR 轮询了。

> “实时”对不同的应用有不同的含义：有些应用要求毫秒级的精确度，而有些应用可能只要几分钟同步一次就够了。要确定最佳的传输方式，首先必须明确自己应用的延迟和性能目标！

15.7.1 通过XHR实现轮询

从服务器取得更新的一个最简单的办法，就是客户端在后台定时发起 XHR 请求，也就是轮询（polling）。如果服务器有新数据，返回新数据，否则返回空响应。

轮询实现起来简单，但也经常效率很低。其中关键在于选择轮询间隔：长轮询间隔意味着延迟交付，而短轮询间隔会导致客户端与服务器间不必要的流量和协议开销。下面看一个简单的例子：

```
function checkUpdates(url) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.onload = function() { ... }; ❶
  xhr.send();
}

setInterval("checkUpdates('/updates'), 60000"); ❷
```

❶ 处理从服务器收到的更新
❷ 每 60 秒发送一个 XHR 请求

- 每个 XHR 请求都是一次独立的 HTTP 请求，平均算下来，每个 HTTP 会带有大约 800 字节的请求 / 响应首部（不算 HTTP cookie）。
- 定时检查很有效，前提是数据能按时到达。可惜的是，按时到达只是例外，而非常规情形。于是，定时轮询也会导致服务器端消息可用与交付给客户端之间的额外延迟。
- 除非考虑周到，否则轮询对于无线网络来说常常会变成性能杀手（参见 8.2 节“消除周期性及无效的数据传输”）。唤醒无线电模块会消耗很多电量！

最佳的轮询间隔是多少？没有唯一的答案。轮询频率取决于应用的需要，而且始终都会存在关于效率和消息延迟的权衡。所以说，轮询最适合间隔时间长，新事件到达时间有规律，且传输数据量大的场景。这个组合可以抵消多余的 HTTP 开销，并将消息交付的延迟最小化。

XHR 轮询的性能建模

为说明如何权衡 XHR 轮询的延迟与开销，我们可以拿一个简单的邮件应用作例子。这个应用使用 XHR 轮询检查服务器上的新邮件。实现步骤如下：

- 客户端每 60 s 发送一次 XHR 检查更新；
- 每次 XHR 请求都包含客户端知道的最近消息 ID；
- 服务器用客户端发送的 ID 查询其消息列表；
- 服务器响应新消息列表或空列表（表示没有更新）。

消息的平均延迟是多长时间？

如果服务器端消息恰好在客户端检查更新之前到达，那么延迟时间最短，只有客户端到服务器的网络延迟。相反，同一条新消息如果在客户端刚刚检查更新之后才到达，那么这条消息就要等到下次检查时才能发送（多等 60 s）。实际上，如果消息是随机到达的，那么每条消息平均要在服务器上等待 30 s 才能让客户端检查到。

轮询的开销有多大？

HTTP 1.x 的请求与响应会增加大约 800 字节的载荷（参见 11.5 节“度量和控制协议开销”）。加上客户端登录，还会有额外的认证 cookie 和消息 ID，假设又会增加 50 字节。那么，返回空消息列表的请求的开销为 850 字节！假设有 10 000 个客户端，全部以 60 s 为间隔向服务器轮询：

$$\left(\frac{850\text{ 字节 } \times 8 \text{ 位} \times 10\,000}{60 \text{ s}}\right) \approx 1.13 \text{ Mbit/s}$$

算下来服务器每秒要处理 167 个请求，每个客户端每次请求要发送 850 字节数据，相当于服务器始终要承担 1.13 Mbit/s 的入站流量！而且，这还是不向客户端发送任何新消息情况下的一个固定速率。

30 s 的延迟是否太高了？那就缩短间隔，但这样一来，就会导致更高的吞吐量和开销。同样还是 10 000 个客户端，如果轮询间隔缩短到 1 s，就会导致 60 Mbit/s 以上的吞吐量！简单来说，轮询间隔越短，代价越大。

15.7.2 通过XHR实现长轮询

定时轮询的一个大问题就是很可能造成大量没必要的空检查。记住这一点之后，让我们来对这个轮询过程做一改进（图 15-1）：在没有更新的时候不再返回空响应，而是把连接保持到有更新的时候。

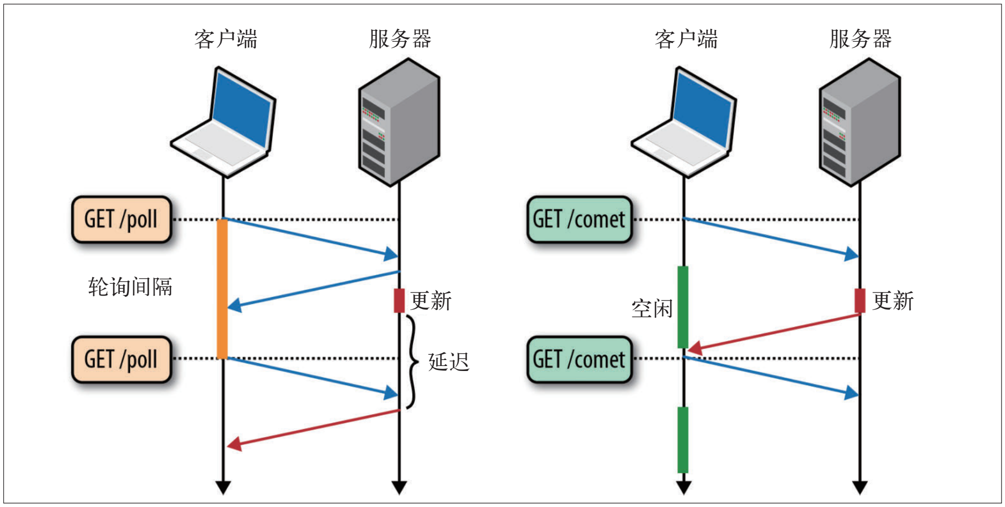


图 15-1：轮询（左）与长轮询（右）的迟延

> 利用长时间保留的 HTTP 请求（“挂起的 GET”）来让服务器向浏览器推送数据的技术，经常被称作 Comet。不过，有时候也有人用其他名字称呼这种技术，比如“保留 AJAX”、“AJAX 推送”或“HTTP 推送”。

通过将连接一直保持打开到有更新（长轮询），就可以把更新立即从服务器发送给客户端。这样，长轮询就解决了消息交付延迟的问题，同时也消灭了空检查，减少了 XHR 请求次数和轮询的整体开销。在交付更新后，长轮询请求完成，然后客户端再发送下一次长轮询请求，等待下一次更新：

```
function checkUpdates(url) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.onload = function() { ❶
    ...
    checkUpdates('/updates'); ❷
  };
  xhr.send();
}

checkUpdates('/updates'); ❸
```

❶ 处理更新并打开新的长轮询 XHR
❷ 发送长轮询请求并等待下次更新（如此不停循环）
❸ 发送第一次长轮询 XHR 请求

这样的话，是不是可以说长轮询永远都比定时轮询好呢？除非更新到达频率已知且

固定，否则长轮询的延迟总是最短的。如果延迟是一个重要考虑因素，那么长轮询就是最好方案。

从另一方面看，还需要更仔细地分析一下开销。首先，每次更新都会伴有相同的 HTTP 开销，即每次更新都需要一次独立的 HTTP 请求。如果更新频率很高，那么长轮询又会导致比定时轮询更多的 XHR 请求！

长轮询通过最小化延迟可以动态适应更新频率，这种行为可能是也可能不是我们所期望的。如果应用可以容许一定时间的延迟，那么定时轮询可能更有效。因为在更新频率很高的情况下，定时轮询就是一个简单的“更新累积”机制，不仅能减少请求次数，还能减少对手机电量的消耗。

> 实践中，并不是所有更新都具有相同的优先级或者延迟要求。因此，你可以考虑采取混合策略：在服务器上累积低优先级的更新，同时对高优先级消息则立即触发更新（参见 8.2 节的“内格尔及有效的服务器推送”）。

Facebook 的 Chat 使用了 XHR 长轮询

实际应用中，长轮询已经成为通过 XHR 交付实时通知的一个使用最广泛的方法。虽然这种方法不一定最有效，但它简单、可靠，在任何支持 XHR 的浏览器中通用。Facebook 流行的 Chat 应用就在 2008 年率先采用了这个简单的方法：

> 为了让一个用户从另一个用户那里取得消息，我们采用了这个方法：在每个 Facebook 页面中加载一个 iframe，让这个 iframe 中的 JavaScript 发送一个 HTTP GET 请求，并保持连接打开，直到服务器把数据返回客户端再关闭连接。如果连接中断或超时，则重新建立连接。这个方法并没有使用任何新技术，它只是 Comet（尤其是 XHR 长轮询）及 BOSH 的一种新用法。
>
> ——Facebook Chat
> Facebook Engineering Blog

今天，通过 Server-Sent Events 和 WebSocket 可以更有效地实现同样的功能。即使如此，XHR 仍然是很多实时通信框架的常用备选方案。如果所有技术都不能用，那么只有让长轮询顶上了！

15.8 XHR使用场景及性能

XMLHttpRequest 是我们从在浏览器中做网页转向开发 Web 应用的关键。首先，它让我们在浏览器中实现了异步通信，但同样重要的是，它还把这个过程变得非常简

单。分派和控制 HTTP 请求只要几行 JavaScript 代码，而其他复杂的任务浏览器都包办了：

- 浏览器格式化 HTTP 请求并解析响应；
- 浏览器强制施加相关的安全（同源）策略；
- 浏览器处理内容协商（如 gzip 压缩）；
- 浏览器处理请求和响应的缓存；
- 浏览器处理认证、重定向……

正因为如此，XHR 成了所有使用 HTTP 请求 / 响应模式来传输数据的一种全能又高性能的机制。想取得需要认证的资源，且传输期间需要压缩，取得后需要缓存以备后用？浏览器会替我们完成这一切，以及更多。作为开发者，我们只要关注自己应用的逻辑即可！

不过，XHR 也有其自身的局限性。如前所述，XHR 标准从未考虑流式数据处理的场景，对它的支持也有限。这就造成了通过 XHR 进行流式数据传输既低效，又不方便。不同的浏览器还有不同的行为，有效的二进制流式传输是不可能的。简言之，XHR 不适合流式数据处理。

类似地，也没有最好的方式通过 XHR 实时交付更新。定时轮询会导致高开销及更新延迟。长轮询的延迟低，但每次更新仍然有开销，因为每次更新都需要一次 HTTP 请求。既想低延迟，又想低开销，除非你有 XHR 流！

于是，尽管 XHR 是一种“实时”交付的流行机制，但从性能角度说，它或许并不是最佳选择。现代浏览器支持比它更简单也更高效的 API，比如 Server-Sent Events 和 WebSocket。事实上，除非你有特别的理由要使用 XHR 轮询，否则应该使用这些新技术。

第16章

服务器发送事件

Server-Sent Events（SSE）让服务器可以向客户端流式发送文本消息，比如服务器上生成的实时通知或更新。为达到这个目标，SSE 设计了两个组件：浏览器中的 `EventSource` 和新的“事件流”数据格式。其中，`EventSource` 可以让客户端以 DOM 事件的形式接收到服务器推送的通知，而新数据格式则用于交付每一次更新。

`EventSource` API 和定义完善的事件流数据格式，使得 SSE 成为了在浏览器中处理实时数据的高效而不可或缺的工具：

- 通过一个长连接低延迟交付；
- 高效的浏览器消息解析，不会出现无限缓冲；
- 自动跟踪最后看到的消息及自动重新连接；
- 消息通知在客户端以 DOM 事件形式呈现。

实际上，SSE 提供的是一个高效、跨浏览器的 XHR 流实现，消息交付只使用一个长 HTTP 连接。然而，与我们自己实现 XHR 流不同，浏览器会帮我们管理连接、解析消息，从而让我们只关注业务逻辑。一句话，SSE 让处理实时数据变得简单高效！接下来我们就揭开它神秘的面纱一探究竟。

16.1 EventSource API

`EventSource` 接口通过一个简单的浏览器 API 隐藏了所有的底层细节，包括建立连接和解析消息。要使用它，只需指定 SSE 事件流资源的 URL，并在该对象上注册相应

的 JavaScript 事件监听器即可：

```
var source = new EventSource("/path/to/stream-url"); ❶

source.onopen = function () { ... }; ❷
source.onerror = function () { ... }; ❸

source.addEventListener("foo", function (event) { ❹
  processFoo(event.data);
});

source.onmessage = function (event) {  ❺
  log_message(event.id, event.data);

  if (event.id== "CLOSE") {
    source.close(); ❻
  }
}
```

❶ 打开到流终点的 SSE 连接
❷ 可选的回调，建立连接时调用
❸ 可选的回调，连接失败时调用
❹ 监听 "foo" 事件，调用自定义代码
❺ 监听所有事件，不明确指定事件类型
❻ 如果服务器发送 "CLOSE" 消息 ID，关闭 SSE 连接

> EventSource 可以像常规 XHR 一样利用 CORS 许可及选择同意机制，实现客户端到远程服务器的流式事件数据传输。

这就是客户端 API 的全部。浏览器会帮我们处理一切：替我们协商建立连接，接收并递增地解析数据，标识消息的范围，最终触发 DOM 事件。EventSource 接口的设计就是让开发者只关注业务逻辑：打开新连接、处理接收到的事件通知，然后在完事儿时终止流。

> SSE 实现了节省内存的 XHR 流。与原始的 XHR 流在连接关闭前会缓冲接收到的所有响应不同，SSE 连接会丢弃已经处理过的消息，而不会在内存中累积。

值得一提的是，EventSource 接口还能自动重新连接并跟踪最近接收的消息：如果连接断开了，EventSource 会自动重新连接到服务器，还可以向服务器发送上一次接收到的消息 ID，以便服务器重传丢失的消息并恢复流。

浏览器怎么知道每个消息的 ID、类型和范围呢？这里就用到了事件流协议。我们之所以可以把繁重的工作托付给浏览器，就是因为这个简单的客户端 API 和定义完善的数据格式。这两个组件总是密切协同，使得浏览器中的应用完全不必理会底层数据协议。

使用自编 JavaScript 模拟 EventSource

SSE 最初是 HTML5 规范的补充，得到了大多数现代浏览器的原生支持。到本书翻译时为止，只有 IE 和 Opera Mini 不支持它。最新状态请参考这里：caniuse.com/eventsource。

不过，好在 EventSource 接口真的很简单，甚至都可以在不支持它的浏览器中使用 JavaScript 库（比如某个“腻子脚本”）来模拟它。类似地，事件流的交付则可以在现有的 XHR 机制基础上实现：

```
if (!window.EventSource) {
  // 加载 JavaScript 腻子脚本
}


var source = new EventSource("/event-stream-endpoint");
...
```

使用腻子脚本的好处，仍然是让我们只关注应用逻辑，而不是因浏览器支持情况闹心。话虽如此，腻子脚本只是提供了一致的 API，底层的 XHR 传输机制依旧不那么高效：

- XHR 轮询会导致消息延迟和很高的请求开销；
- XHR 长轮询能最小化延迟，但开销还是很高；
- XHR 对流的支持有限，且在内存中缓冲所有数据。

在对高效 XHR 流没有原生支持的时候，腻子脚本可以用轮询、长轮询或 XHR 流代替，这几种方案各有利弊。要了解详细信息，请参考 15.7 节“实时通知与交付”。

简单来讲，你得自己检查一下选择的腻子脚本，保证它能达到你的性能要求。很多流行的库（比如 `jQuery.EventSource`）都使用 XHR 轮询模拟 SSE，为我们提供了简单却不那么高效的选择。

16.2　Event Stream协议

SSE 事件流是以流式 HTTP 响应形式交付的：客户端发起常规 HTTP 请求，服务器以自定义的“text/event-stream”内容类型响应，然后交付 UTF-8 编码的事件数据。这么简单几句话似乎都有点说复杂了，看一个例子：

```
=> 请求
GET /stream HTTP/1.1 ❶
Host: example.com
Accept: text/event-stream

<= 响应
HTTP/1.1 200 OK ❷
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 15000 ❸

data: First message is a simple string. ❹

data: {"message": "JSON payload"} ❺

event: foo ❻
data: Message of type "foo"

id: 42 ❼
event: bar
data: Multi-line message of
data: type "bar" and id "42"

id: 43 ❽
data: Last message, id "43"
```

❶ 客户端通过 `EventSource` 接口发起连接
❷ 服务器以 `"text/event-stream"` 内容类型响应
❸ 服务器设置连接中断后重新连接的间隔时间（15 s）
❹ 不带消息类型的简单文本事件
❺ 不带消息类型的 JSON 数据载荷
❻ 类型为 `"foo"` 的简单文本事件
❼ 带消息 ID 和类型的多行事件
❽ 带可选 ID 的简单文本事件

以上事件流协议很好理解，也很好实现：

- 事件载荷就是一或多个相邻 `data` 字段的值；
- 事件可以带 ID 和 `event` 表示事件类型；
- 事件边界用换行符标识。

在接收端，`EventSource` 接口通过检查换行分隔符来解析到来的数据流，从 `data` 字

段中提取有效载荷，检查可选的 ID 和类型，最后再分派一个 DOM 事件告知应用。如果存在某个类型，那么就会触发自定义的 DOM 事件处理程序；否则，就会调用通用的 `onmessage` 回调，参见 16.1 节“EventSource API”。

SSE 中的 UTF-8 编码与二进制传输

`EventSource` 不会对实际载荷进行任何额外处理：从一或多个 `data` 字段中提取出来的消息，会被拼接起来直接交给应用。因此，服务器可以推送任何文本格式（例如，简单字符串、JSON，等等），应用必须自己解码。

话虽如此，但所有事件源数据都是 UTF-8 编码的：SSE 不是为传输二进制载荷而设计的！如果有必要，可以把二进制对象编码为 base64 形式，然后再使用 SSE。但这样会导致很高（33%）的字节开销，参见 11.7 节“嵌入资源”。

担心 UTF-8 编码也会造成高开销？ SSE 连接本质上是 HTTP 流式响应，因此响应是可以压缩的（如 gzip 压缩），就跟压缩其他 HTTP 响应一样，而且是动态压缩！虽然 SSE 不是为传输二进制数据而设计的，但它却是一个高效的机制——只要让你的服务器对 SSE 流应用 gzip 压缩。

不支持二进制传输是有意为之的。SSE 的设计目标是简单、高效，作为一种服务器向客户端传送文本数据的机制。如果你想传输二进制数据，WebSocket 才是更合适的选择。

最后，除了自动解析事件数据，SSE 还内置支持断线重连，以及恢复客户端因断线而丢失的消息。默认情况下，如果连接中断，浏览器会自动重新连接。SSE 规范建议的间隔时间是 2~3 s，这也是大多数浏览器采用的默认值。不过，服务器也可以设置一个自定义的间隔时间，只要在推送任何消息时向客户端发送一个 `retry` 命令即可。

类似地，服务器还可以给每条消息关联任意 ID 字符串。浏览器会自动记录最后一次收到的消息 ID，并在发送重连请求时自动在 HTTP 首部追加“Last-Event-ID”值。下面看一个例子：

```
(既有 SSE 连接)
retry: 4500 ❶

id: 43 ❷
data: Lorem ipsum

(连接断开)
(4500 ms 后)

=> 请求
```

```
GET /stream HTTP/1.1 ❸
Host: example.com
Accept: text/event-stream
Last-Event-ID: 43

<= 响应
HTTP/1.1 200 OK ❹
Content-Type: text/event-stream
Connection: keep-alive
Transfer-Encoding: chunked

id: 44 ❺
data: dolor sit amet
```

❶ 服务器将客户端的重连间隔设置为 4.5 s
❷ 简单文本事件，ID:43
❸ 带最后一次事件 ID 的客户端重连请求
❹ 服务器以 `'text/event-stream'` 内容类型响应
❺ 简单文本事件，ID:44

客户端应用不必为重新连接和记录上一次事件 ID 编写任何代码。这些都由浏览器自动完成，然后就是服务器负责恢复了。值得注意的是，根据应用的要求和数据流，服务器可以采取不同的实现策略。

- 如果丢失消息可以接受，就不需要事件 ID 或特殊逻辑，只要让客户端重连并恢复数据流即可。
- 如果必须恢复消息，那服务器就需要指定相关事件的 ID，以便客户端在重连时报告最后接收到的 ID。同样，服务器也需要实现某种形式的本地缓存，以便恢复并向客户端重传错过的消息。

当然，像要保留多少条消息这种细节一定取决于具体的应用。另外，要知道 ID 是可选的事件流字段。而服务器也可以在交付的事件流中对特定消息设置检查点或者里程碑标记。一句话，根据你的需求，实现服务器逻辑。

16.3 SSE使用场景及性能

SSE 是服务器向客户端发送实时文本消息的高性能机制：服务器可以在消息刚刚生成就将其推送到客户端（低延迟），使用长连接的事件流协议，而且可以 gzip 压缩（低开销），浏览器负责解析消息，也没有无限缓冲。再加上超级简单的 EventSource API 能自动重新连接和把消息通知作为 DOM 事件，使得 SSE 成为处理实时数据不可或缺的得力工具！

SSE 主要有两个局限。一，只能从服务器向客户端发送数据，不能满足需要请求流的场景（比如向服务器流式上传大文件）；二，事件流协议设计为只能传输 UTF-8 数据，即使可以传输二进制流，效率也不高。

话虽如此，UTF-8 的限制往往可以在应用层克服：SSE 可以通知应用说服务器上有一个新的二进制文件可以下载了，应用只要再分派一个 XHR 请求去下载即可。虽然这样多了一次往返延迟，但也能利用上 XHR 提供的诸多便利：响应缓存、传输编码（压缩），等等。如果文件是流式下载的，那它就无法被浏览器缓存。

> 实时推送就像轮询一样，可能会极大影响电池的待机时间。首先，可以考虑批量处理消息，尽量少唤醒无线电模块。其次，避免不必要的长连接，SSE 连接在无线电空闲时不会断开。更多信息，请参考 8.2 节“消除周期性及无效的数据传输”。

通过 TLS 实现 SSE 流

SSE 通过常规 HTTP 连接实现了简单便捷的实时传输机制，服务器端容易部署，客户端也容易打补丁。可是，现有网络中间设备，比如代理服务器和防火墙，都不支持 SSE，而这有可能带来问题：中间设备可能会缓冲事件流数据，导致额外延迟，甚至彻底毁掉 SSE 连接。

如果你碰到了这样或类似的问题，那么可以考虑通过 TLS 发送 SSE 事件流，具体请参考 4.1 节中的“Web 代理、中间设备、TLS 与新协议”。

第17章

WebSocket

WebSocket 可以实现客户端与服务器间双向、基于消息的文本或二进制数据传输。它是浏览器中最靠近套接字的 API。但 WebSocket 连接远远不是一个网络套接字，因为浏览器在这个简单的 API 之后隐藏了所有的复杂性，而且还提供了更多服务：

- 连接协商和同源策略；
- 与既有 HTTP 基础设施的互操作；
- 基于消息的通信和高效消息分帧；
- 子协议协商及可扩展能力。

WebSocket 是浏览器中最通用最灵活的一个传输机制，其极简的 API 可以让我们在客户端和服务器之间以数据流的形式实现各种应用数据交换（包括 JSON 及自定义的二进制消息格式），而且两端都可以随时向另一端发送数据。

不过，自定义数据交换协议的问题通常也在于自定义。因为应用必须考虑状态管理、压缩、缓存及其他原来由浏览器提供的服务。设计限制和性能权衡始终会有，利用 WebSocket 也不例外。简单来说，WebSocket 并不能取代 HTTP、XHR 或 SSE，而为了追求最佳性能，关键还是要利用这些机制的长处。

> WebSocket 由多个标准构成：WebSocket API 是 W3C 定义的，而 WebSocket 协议（RFC 6455）及其扩展则由 HyBi Working Group（IETF）定义。

17.1 WebSocket API

浏览器提供的 WebSocket API 可谓简约。当然，简约背后隐藏着连接管理和消息处理等底层细节。发起新连接，需要 WebSocket 资源的 URL 和一些应用回调：

```
var ws = new WebSocket('wss://example.com/socket'); ❶

ws.onerror = function (error) { ... } ❷
ws.onclose = function () { ... } ❸

ws.onopen = function () { ❹
  ws.send("Connection established. Hello server!"); ❺
}

ws.onmessage = function(msg) { ❻
  if(msg.data instanceof Blob) { ❼
    processBlob(msg.data);
  } else {
    processText(msg.data);
  }
}
```

❶ 打开新的安全 WebSocket 连接（wss）

❷ 可选的回调，在连接出错时调用

❸ 可选的回调，在连接终止时调用

❹ 可选的回调，在 WebSocket 连接建立时调用

❺ 客户端先向服务器发送一条消息

❻ 回调函数，服务器每发回一条消息就调用一次

❼ 根据接收到的消息，决定调用二进制还是文本处理逻辑

这个 API 非常直观。事实上，应该说它与上一章介绍的 EventSource API 很像。这是故意这么设计的，因为 WebSocket 也提供类似和扩展的功能。当然，除了相似性之外，还有很多重要的差别。下面我们就逐个介绍。

模拟 WebSocket

WebSocket 已经经历了很多次版本升级、实现回滚和安全考验。好消息是，RFC 6455 定义的最新版本（v13）已经得到所有现代浏览器支持。唯一尚未支持它的是 Opera mini，可参见 caniuse.com/websockets。

与使用腻子脚本模拟 SSE 类似（参见 16.1 节的“使用自编 JavaScript 模拟 EventSource”），WebSocket 也可以使用 JavaScript 库来模拟。不过，模拟 WebSocket 最难的地方不是 API，而是传输层！因此，选择的腻子脚本及其后备机制（XHR、轮询、EventSource、iframe 轮询，等等）对模拟方案的性能会有很大影响。

为简化跨浏览器部署，SockJS 等流行的库都提供了类似 WebSocket 对象的实现，并更进一步实现了支持 WebSocket 的自定义服务器，以及各种备用传输方案。自定义服务器和客户端构成了所谓的“无缝备用”：性能虽然差强人意，但应用的 API 不变。

其他库，比如 Socket.IO 走得甚至更远，除提供多套后备传输机制，还实现了心跳检测、超时、自动重连等功能。

在选择 Socket.IO 这样的腻子脚本或“实时框架”时，一定要留心其底层实现，以及客户端和服务器的配置：保证尽可能利用原生 WebSocket 接口以求最佳性能，然后确保备用传输机制能满足你的性能要求。

17.1.1 WS与WSS

WebSocket 资源 URL 采用了自定义模式：ws 表示纯文本通信（如 ws://example.com/socket），wss 表示使用加密信道通信（TCP+TLS）。为什么不使用 http 而要自定义呢？

WebSocket 的主要目的，是在浏览器中的应用与服务器之间提供优化的、双向通信机制。可是，WebSocket 的连接协议也可以用于浏览器之外的场景，可以通过非 HTTP 协商机制交换数据。考虑到这一点，HyBi Working Group 就选择采用了自定义的 URL 模式。

使用自定义的 URL 模式虽然让非 HTTP 协商成为可能，但实践中还没有既定标准可以作为建立 WebSocket 会话的替代握手机制。

17.1.2 接收文本和二进制数据

WebSocket 通信只涉及消息，应用代码无需担心缓冲、解析、重建接收到的数据。比如，服务器发来了一个 1 MB 的净荷，应用的 onmessage 回调只会在客户端接收到全部数据时才会被调用。

此外，WebSocket 协议不作格式假设，对应用的净荷也没有限制：文本或者二进制数据都没问题。从内部看，协议只关注消息的两个信息：净荷长度和类型（前者是一个可变长度字段），据以区别 UTF-8 数据和二进制数据。

浏览器接收到新消息后，如果是文本数据，会自动将其转换成 DOMString 对象，如果是二进制数据或 Blob 对象，会直接将其转交给应用。唯一可以（作为性能暗示和优化措施）多余设置的，就是告诉浏览器把接收到的二进制数据转换成 ArrayBuffer 而非 Blob：

```
var ws = new WebSocket('wss://example.com/socket');
ws.binaryType = "arraybuffer"; ❶

ws.onmessage = function(msg) {
  if(msg.data instanceof ArrayBuffer) {
    processArrayBuffer(msg.data);
  } else {
    processText(msg.data);
  }
}
```

❶ 如果接收到二进制数据，将其强制转换成 `ArrayBuffer`

> 用户代理可以将这个选项看作一个暗示，以决定如何处理接收到的二进制数据：如果这里设置为“blob”，那就可以放心地将其转存到磁盘上；而如果设置为“arraybuffer”，那很可能在内存里处理它更有效。自然地，我们鼓励用户代理使用更细微的线索，以决定是否将到来的数据放到内存里……
>
> ——The WebSocket API
> W3C Candidate Recommendation

Blob 对象一般代表一个不可变的文件对象或原始数据。如果你不需要修改它或者不需要把它切分成更小的块，那这种格式是理想的（比如，可以把一个完整的 Blob 对象传给 `img` 标签，参见 15.3 节“通过 XHR 下载数据”）。而如果你还需要再处理接收到的二进制数据，那么选择 `ArrayBuffer` 应该更合适。

使用 JavaScript 解码二进制数据

`ArrayBuffer` 表示一个普通的、固定长度的二进制数据缓冲。不过，可以用 `ArrayBuffer` 创建一或多个 `ArrayBufferView` 对象，每一个都可以通过特定的格式来展示缓冲中的内容。比如，假设我们需要处理下面类似 C 的二进制数据结构：

```
struct someStruct {
  char username[16];
  unsigned short id;
  float scores[32];
};
```

在取得这个类型的 ArrayBuffer 对象后，可以对同一个缓冲创建多个不同的视图，每个视图的偏移量和数据类型都可以不一样：

```
var buffer = msg.data;

var usernameView = new Uint8Array(buffer, 0, 16);
var idView = new Uint16Array(buffer, 16, 1);
var scoresView = new Float32Array(buffer, 18, 32);

console.log("ID: " + idView[0] + " username: " + usernameView[0]);
for (var j = 0; j < 32; j++) { console.log(scoresView[j]) }
```

每个视图都以父缓冲、开始字节偏移量和要处理的元素数作为参数，其中偏移量根据之前字段的大小计算。结果，ArrayBuffer 和 WebSocket 实际上为我们在浏览器中处理二进制数据提供了所有必要的工具。

17.1.3 发送文本和二进制数据

建立了 WebSocket 连接后，客户端就可以随时发送或接收 UTF-8 或二进制消息。WebSocket 提供的是一条双向通信的信道，也就是说，在同一个 TCP 连接上，可以双向传输数据：

```
var ws = new WebSocket('wss://example.com/socket');

ws.onopen = function () {
  socket.send("Hello server!"); ❶

  socket.send(JSON.stringify({'msg': 'payload'})); ❷


  var buffer = new ArrayBuffer(128);
  socket.send(buffer); ❸


  var intview = new Uint32Array(buffer);
  socket.send(intview); ❹


  var blob = new Blob([buffer]);
  socket.send(blob); ❺
}
```

❶ 发送 UTF-8 编码的文本消息
❷ 发送 UTF-8 编码的 JSON 净荷
❸ 发送二进制 ArrayBuffer
❹ 发送二进制 ArrayBufferView
❺ 发送二进制 Blob

WebSocket API 可以接收 UTF-8 编码的 DOMString 对象，也可以接收 ArrayBuffer、ArrayBufferView 或 Blob 等二进制数据。但要注意，所有二进制数据类型只是为了简化 API：在传输中，只通过一位（bit）即可将 WebSocket 帧标记为二进制或者文本。假如应用或服务器需要传输其他的内容类型，就必须通过其他机制来沟通这个信息。

这里的 send() 方法是异步的：提供的数据会在客户端排队，而函数则立即返回。特别是在传输大文件的时候，千万别因为返回快，就错误地以为数据已经发送出去了！要监控在浏览器中排队的数据量，可以查询套接字的 bufferedAmount 属性：

```
var ws = new WebSocket('wss://example.com/socket');

ws.onopen = function () {
  subscribeToApplicationUpdates(function(evt) { ❶
    if (ws.bufferedAmount == 0) ❷
      ws.send(evt.data); ❸
  });
};
```

❶ 预订应用更新（如游戏状态更新）

❷ 检查客户端缓冲的数据量

❸ 如果缓冲是空的，发送下一次更新

前面的例子是要向服务器发送应用数据，但前提是客户端缓冲区已经没有之前待发送的数据了。为什么非要做这个检查？所有 WebSocket 消息都会按照它们在客户端排队的次序逐个发送。因此，大量排队的消息，甚至一个大消息，都可能导致排在它后面的消息延迟——队首阻塞！

为解决这个问题，应用可以将大消息切分成小块，通过监控 `bufferedAmount` 的值来避免队首阻塞。甚至还可以实现自己的优先队列，而不是盲目都把它们送到套接字上排队。

> 很多应用都会生成多种消息：有高优先级的更新，也有低优先级的更新。前者比如流量控制消息，后者比如后台传输。要实现最优化传输，应用必须关心任意时刻在套接字上排队的是什么消息！

17.1.4　子协议协商

WebSocket 协议对每条消息的格式事先不作任何假设：仅用一位标记消息是文本还是二进制，以便客户端和服务器有效地解码数据，而除此之外的消息内容就是未知的。

此外，与 HTTP 或 XHR 请求不同——它们是通过每次请求和响应的 HTTP 首部来沟通元数据，WebSocket 并没有等价的机制。因此，如果需要沟通关于消息的元数据，客户端和服务器必须达成沟通这一数据的子协议。

- 客户端和服务器可以提前确定一种固定的消息格式，比如所有通信都通过 JSON 编码的消息或者某种自定义的二进制格式进行，而必要的元数据作为这种数据结构的一个部分。
- 如果客户端和服务器要发送不同的数据类型，那它们可以确定一个双方都知道的消息首部，利用它来沟通说明信息或有关净荷的其他解码信息。

- 混合使用文本和二进制消息可以沟通净荷和元数据，比如用文本消息实现 HTTP 首部的功能，后跟包含应用净荷的二进制消息。

以上只列举了几种可能的策略。与 WebSocket 消息的灵活性和低延迟对应的，就是应用逻辑必须复杂一点。不过，消息的串行化和元数据管理只是问题的一方面！确定了消息的串行格式化，怎么保证客户端和服务器相互理解，怎么确保它们同步呢？

好在，WebSocket 为此提供了一个简单便捷的子协议协商 API。客户端可以在初次连接握手时，告诉服务器自己支持哪种协议：

```
var ws = new WebSocket('wss://example.com/socket',
                       ['appProtocol', 'appProtocol-v2']); ❶

ws.onopen = function () {
  if (ws.protocol == 'appProtocol-v2') { ❷
    ...
  } else {
    ...
  }
}
```

❶ 在 `WebSocket` 握手期间发送子协议数组
❷ 检查服务器选择了哪个子协议

如这个例子所示，`WebSocket` 构造函数可以接受一个可选的子协议名字的数组，通过这个数组，客户端可以向服务器通告自己能够理解或希望服务器接受的协议。这个协议数组会发送给服务器，服务器可以从中挑选一个。

如果子协议协商成功，就会触发客户端的 `onopen` 回调，应用可以查询 `WebSocket` 对象上的 `protocol` 属性，从而得知服务器选定的协议。另一方面，服务器如果不支持客户端声明的任何一个协议，则 `WebSocket` 握手是不完整的，此时会触发 `onerror` 回调，连接断开。

> 子协议名由应用自己定义，且在初次 HTTP 握手期间发送给服务器。除此之外，指定的子协议对核心 `WebSocket` API 不会有任何影响。

17.2　WebSocket协议

HyBi Working Group 制定的 WebSocket 通信协议（RFC 6455）包含两个高层组件：开放性 HTTP 握手用于协商连接参数，二进制消息分帧机制用于支持低开销的基于

消息的文本和二进制数据传输。

> WebSocket 协议尝试在既有 HTTP 基础设施中实现双向 HTTP 通信，因此也使用 HTTP 的 80 和 443 端口……不过，这个设计不限于通过 HTTP 实现 WebSocket 通信，未来的实现可以在某个专用端口上使用更简单的握手，而不必重新定义么一个协议。
>
> ——WebSocket Protocol
> RFC 6455

WebSocket 协议是一个独立完善的协议，可以在浏览器之外实现。不过，它的主要应用目标还是实现浏览器应用的双向通信。

17.2.1　二进制分帧层

客户端和服务器 WebSocket 应用通过基于消息的 API 通信：发送端提供任意 UTF-8 或二进制的净荷，接收端在整个消息可用时收到通知。为此，WebSocket 使用了自定义的二进制分帧格式（图 17-1），把每个应用消息切分成一或多个帧，发送到目的地之后再组装起来，等到接收到完整的消息后再通知接收端。

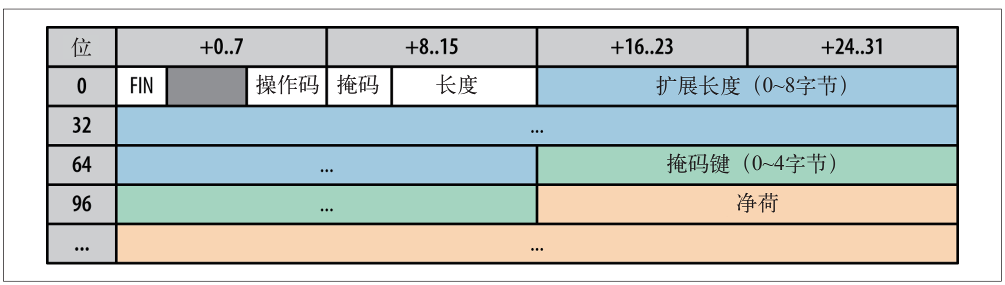


图 17-1：WebSocket 帧格式：2~14 字节 + 净荷

- 帧
  最小的通信单位，包含可变长度的帧首部和净荷部分，净荷可能包含完整或部分应用消息。
- 消息
  一系列帧，与应用消息对等。

是否把消息分帧由客户端和服务器实现决定。事实上，应用可以对个别 WebSocket 帧或如何分帧毫无概念。即便如此，理解每个 WebSocket 帧也很重要。

- 每一帧的第一位（FIN）表示当前帧是不是消息的最后一帧。一条消息有可能只对应一帧。

- 操作码（4 位）表示被传输帧的类型：传输应用数据时，是文本（1）还是二进制（2）；连接有效性检查时，是关闭（8）、呼叫（ping，9）还是回应（pong，10）。
- 掩码位表示净荷是否有掩码（只适用于客户端发送给服务器的消息）。
- 净荷长度由可变长度字段表示：
  - 如果是 0~125，就是净荷长度；
  - 如果是 126，则接下来 2 字节表示的 16 位无符号整数才是这一帧的长度；
  - 如果是 127，则接下来 8 字节表示的 64 位无符号整数才是这一帧的长度。
- 掩码键包含 32 位值，用于给净荷加掩护。
- 净荷包含应用数据，如果客户端和服务器在建立连接时协商过，也可以包含自定义的扩展数据。

> 所有客户端发送帧的净荷都要使用帧首部中指定的值加掩码，这样可以防止客户端中运行的恶意脚本对不支持 WebSocket 的中间设备进行缓存投毒攻击（cache poisoning attack）。要了解这种攻击的细节，请参考 W2SP 2011 的论文“Talking to Yourself for Fun and Profit”（http://w2spconf.com/2011/papers/websocket.pdf）。

算下来，服务器发送的每个 WebSocket 帧会产生 2~10 字节的分帧开销。而客户端必须发送掩码键，这又会增加 4 字节，结果就是 6~14 字节的开销。除此之外，没有其他元数据（比如首部字段或其他关于净荷的信息）：所有 WebSocket 通信都是通过交换帧实现的，而帧将净荷视为不透明的应用数据块。

WebSocket 的多路复用及队首阻塞

WebSocket 很容易发生队首阻塞的情况：消息可能会被分成一或多个帧，但不同消息的帧不能交错发送，因为没有与 HTTP 2.0 分帧机制中“流 ID”对等的字段（参见 12.3.2 节“流、消息和帧”）。

显然，如果一个大消息被分成多个 WebSocket 帧，就会阻塞其他消息的帧。如果你的应用不容许有交付延迟，那可以小心控制每条消息的净荷大小，甚至可以考虑把大消息拆分成多个小消息！

WebSocket 不支持多路复用，还意味着每个 WebSocket 连接都需要一个专门的 TCP 连接。对于 HTTP 1.x 而言，由于浏览器针对每个来源有连接数量限制，因此可能会导致问题（参见 11.3 节中的“消耗客户端和服务器资源”）。

好在，HyBi Working Group 正着手制定的新的“Multiplexing Extension for WebSockets”（WebSockets 多路复用扩展）会解决这个问题：

> 这个扩展通过封装帧并加上信道 ID，可以让一个 TCP 连接支持多个虚拟 WebSocket 连接……这个多路复用扩展维护独立的逻辑信道，每个逻辑信道与独立的 WebSocket 连接没有差别，包括独立的握手首部。
>
> ——WebSocket Multiplexing（Draft 10）

有了这个扩展后，多个 WebSocket 连接（信道）就可能在同一个 TCP 连接上得到复用。可是，每个信道依旧容易产生队首阻塞问题！可能的解决方案是使用不同的信道，或者专用 TCP 连接，多路并行发送消息。

最后，注意前面的扩展仅对 HTTP 1.x 连接是必要的。虽然通过 HTTP 2.0 传输 WebSocket 帧的官方规范尚未发布，但相对来说就容易多了。因为 HTTP 2.0 内置了流的多路复用，只要通过 HTTP 2.0 的分帧机制来封装 WebSocket 帧，多个 WebSocket 连接就可以在一个会话中传输。

17.2.2 协议扩展

WebSocket 规范允许对协议进行扩展：数据格式和 WebSocket 协议的语义可以通过新的操作码和数据字段扩展。虽然有些不同寻常，但这却是一个非常强大的特性，因为它允许客户端和服务器在基本的 WebSocket 分帧层之上实现更多功能，又不需要应用代码介入或协作。

WebSocket 协议扩展有哪些例子？负责制定 WebSocket 规范的 HyBi Working Group 就进行了两项扩展。

- 多路复用扩展（A Multiplexing Extension for WebSockets）
  这个扩展可以将 WebSocket 的逻辑连接独立出来，实现共享底层的 TCP 连接。

- 压缩扩展（Compression Extensions for WebSocket）
  给 WebSocket 协议增加了压缩功能。

如前所述，每个 WebSocket 连接都需要一个专门的 TCP 连接，这样效率很低。多路复用扩展解决了这个问题。它使用“信道 ID”扩展每个 WebSocket 帧，从而实现多个虚拟的 WebSocket 信道共享一个 TCP 连接。

类似地，基本的 WebSocket 规范没有压缩数据的机制或建议，每个帧中的净荷就是应用提供的净荷。虽然这对优化的二进制数据结构不是问题，但除非应用实现自己的压缩和解压缩逻辑，否则很多情况下都会造成传输载荷过大的问题。实际上，压缩扩展就相当于 HTTP 的传输编码协商。

要使用扩展，客户端必须在第一次的 Upgrade 握手中通知服务器，服务器必须选择

并确认要在商定连接中使用的扩展。下面我们就来看一个 Upgrade 的例子。

> **WebSocket 多路复用与压缩在现实中的应用**
>
> 截止到 2013 年年中，WebSocket 的多路复用还没有得到任何主流浏览器支持。类似地，对压缩扩展的支持也很有限：谷歌 Chrome 和最新的 WebKit 浏览器可能会向服务器发送“x-webkit-deflate-frame”扩展。然而这个 deflate-frame 是基于该标准的过时版本实现的，将来可能会被抛弃。
>
> 压缩扩展的早期版本是“逐帧”压缩，这对要分成多个帧的大消息显然不是最佳方法。最新版本已经修改为以消息为单位压缩，这是好消息。坏消息是，以消息为单位压缩还是实验性的，目前尚未得到主流浏览器支持。
>
> 结果，就是应用需要密切关注传输的数据类型，并在可行的情况下采用自己的压缩方案。换句话说，至少在所有主流浏览器原生支持 WebSocket 之前，你要考虑这个方案。这对移动应用尤为重要，因为移动应用中的每个字节都可能给用户带来不必要的成本。

17.2.3 HTTP升级协商

WebSocket 协议提供了很多强大的特性：基于消息的通信、自定义的二进制分帧层、子协议协商、可选的协议扩展，等等。换句话说，在交换数据之前，客户端必须与服务器协商适当的参数以建立连接。

利用 HTTP 完成握手有几个好处。首先，让 WebSockets 与现有 HTTP 基础设施兼容：WebSocket 服务器可以运行在 80 和 443 端口上，这通常是对客户端唯一开放的端口。其次，让我们可以重用并扩展 HTTP 的 Upgrade 流，为其添加自定义的 WebSocket 首部，以完成协商。

- `Sec-WebSocket-Version`
  客户端发送，表示它想使用的 WebSocket 协议版本（“13”表示 RFC 6455）。如果服务器不支持这个版本，必须回应自己支持的版本。

- `Sec-WebSocket-Key`
  客户端发送，自动生成的一个键，作为一个对服务器的“挑战”，以验证服务器支持请求的协议版本。

- `Sec-WebSocket-Accept`
  服务器响应，包含 `Sec-WebSocket-Key` 的签名值，证明它支持请求的协议版本。

- `Sec-WebSocket-Protocol`

  用于协商应用子协议：客户端发送支持的协议列表，服务器必须只回应一个协议名。

- `Sec-WebSocket-Extensions`

  用于协商本次连接要使用的 WebSocket 扩展：客户端发送支持的扩展，服务器通过返回相同的首部确认自己支持一或多个扩展。

有了这些协商字段，就可以在客户端和服务器之间进行 HTTP Upgrade 并协商新的 WebSocket 连接了：

```
GET /socket HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
Connection: Upgrade
Upgrade: websocket ❶
Sec-WebSocket-Version: 13 ❷
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ== ❸
Sec-WebSocket-Protocol: appProtocol, appProtocol-v2 ❹
Sec-WebSocket-Extensions: x-webkit-deflate-message, x-custom-extension ❺
```

❶ 请求升级到 WebSocket 协议
❷ 客户端使用的 WebSocket 协议版本
❸ 自动生成的键，以验证服务器对协议的支持
❹ 可选的应用指定的子协议列表
❺ 可选的客户端支持的协议扩展列表

与浏览器中客户端发起的任何连接一样，WebSocket 请求也必须遵守同源策略：浏览器会自动在升级握手请求中追加 Origin 首部，远程服务器可能使用 CORS 判断接受或拒绝跨源请求 [参见 15.2 节“跨源资源共享（CORS）”]。要完成握手，服务器必须返回一个成功的“Switching Protocols”（切换协议）响应，并确认选择了客户端发送的哪个选项：

```
HTTP/1.1 101 Switching Protocols ❶
Upgrade: websocket
Connection: Upgrade
Access-Control-Allow-Origin: http://example.com ❷
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo= ❸
Sec-WebSocket-Protocol: appProtocol-v2 ❹
Sec-WebSocket-Extensions: x-custom-extension ❺
```

❶ 101 响应码确认升级到 WebSocket 协议
❷ CORS 首部表示选择同意跨源连接
❸ 签名的键值验证协议支持

❹ 服务器选择的应用子协议
❺ 服务器选择的 WebSocket 扩展

> 所有兼容 RFC 6455 的 WebSocket 服务器都使用相同的算法计算客户端挑战的答案：将 `Sec-WebSocket-Key` 的内容与标准定义的唯一 GUID 字符串拼接起来，计算出 SHA1 散列值，结果是一个 base-64 编码的字符串，把这个字符串发给客户端即可。

最低限度，成功的 WebSocket 握手必须是客户端发送协议版本和自动生成的挑战值，服务器返回 101 HTTP 响应码（Switching Protocols）和散列形式的挑战答案，确认选择的协议版本：

- 客户端必须发送 `Sec-WebSocket-Version` 和 `Sec-WebSocket-Key`；
- 服务器必须返回 `Sec-WebSocket-Accept` 确认协议；
- 客户端可以通过 `Sec-WebSocket-Protocol` 发送应用子协议列表；
- 服务器必须选择一个子协议并通过 `Sec-WebSocket-Protocol` 返回协议名；如果服务器不支持任何一个协议，连接断开；
- 客户端可以通过 `Sec-WebSocket-Extensions` 发送协议扩展；
- 服务器可以通过 `Sec-WebSocket-Extensions` 确认一或多个扩展；如果服务器没有返回扩展，则连接不支持扩展。

最后，前述握手完成后，如果握手成功，该连接就可以用作双向通信信道交换 WebSocket 消息。从此以后，客户端与服务器之间不会再发生 HTTP 通信，一切由 WebSocket 协议接管。

代理、中间设备与 WebSocket

实践中，考虑到安全和保密，很多用户都只开放有限的端口，通常只有 80（HTTP）和 443（HTTPS）。正因为如此，WebSocket 协商是通过 HTTP Upgrade 流进行的，这样可以确保与现有网络策略及基础设施兼容。

不过，正如 4.1 节的“Web 代理、中间设备、TLS 与新协议”所说，很多现有的 HTTP 中间设备可能不理解新的 WebSocket 协议，而这可能导致各种问题：盲目的连接升级、意外缓冲 WebSocket 帧、不明就里地修改内容、把 WebSocket 流量误当作不完整的 HTTP 通信，等等。

WebSocket 的 Key 和 Accept 握手可以解决其中一些问题：这是服务器的一个安全策略，而盲目“升级”连接的中间设备可能并不理解 WebSocket 协议。虽然这个预防措施对某些代理可以解决问题，但对于那些“透明代理”还是不行，它们可能会分析并意外地修改数据。

解决之道？建立一条端到端的安全通道。比如，使用 WSS！在执行 HTTP Upgrade 握手之前，先协商一次 TLS 会话，在客户端与服务器之间建立一条加密通道，就可以解决前述所有问题。这个方案尤其适合移动客户端，因为它们的流量经常要穿越各种代理服务，这些代理服务很可能不认识 WebSocket。

17.3 WebSocket使用场景及性能

`WebSocket` API 提供一个简单的接口，能够在客户端与服务器之间实现基于消息的双向通信，可以是文本数据，可以是二进制数据：把 WebSocket URL 传递给构造函数，设置几个 JavaScript 回调函数，就好了——剩下的就全都由浏览器负责了。再加上 WebSocket 协议提供的二进制分帧、可扩展性以及子协议协商，使得 WebSocket 成为在浏览器中采用自定义应用协议的最佳选择。

不过，就跟以前关于性能的讨论一样，虽然 WebSocket 协议的实现复杂性对应用隐藏了，但何时以及如何使用 WebSocket，毋庸置疑会对性能产生巨大影响。WebSocket 不能取代 XHR 或 SSE，而要获得最佳性能，我们必须善于利用它的长处！

请参考 15.8 节“XHR 使用场景及性能”和 16.3 节“SSE 使用场景及性能”，以了解各种传输机制的性能特点。

17.3.1 请求和响应流

WebSocket 是唯一一个能通过同一个 TCP 连接实现双向通信的机制（图 17-2），客户端和服务器随时可以交换数据。因此，WebSocket 在两个方向上都能保证文本和二进制应用数据的低延迟交付。

- XHR 是专门为“事务型”请求 / 响应通信而优化的：客户端向服务器发送完整的、格式良好的 HTTP 请求，服务器返回完整的响应。这里不支持请求流，在 Streams API 可用之前，没有可靠的跨浏览器响应流 API。
- SSE 可以实现服务器到客户端的高效、低延迟的文本数据流：客户端发起 SSE 连接，服务器使用事件源协议将更新流式发送给客户端。客户端在初次握手后，不能向服务器发送任何数据。

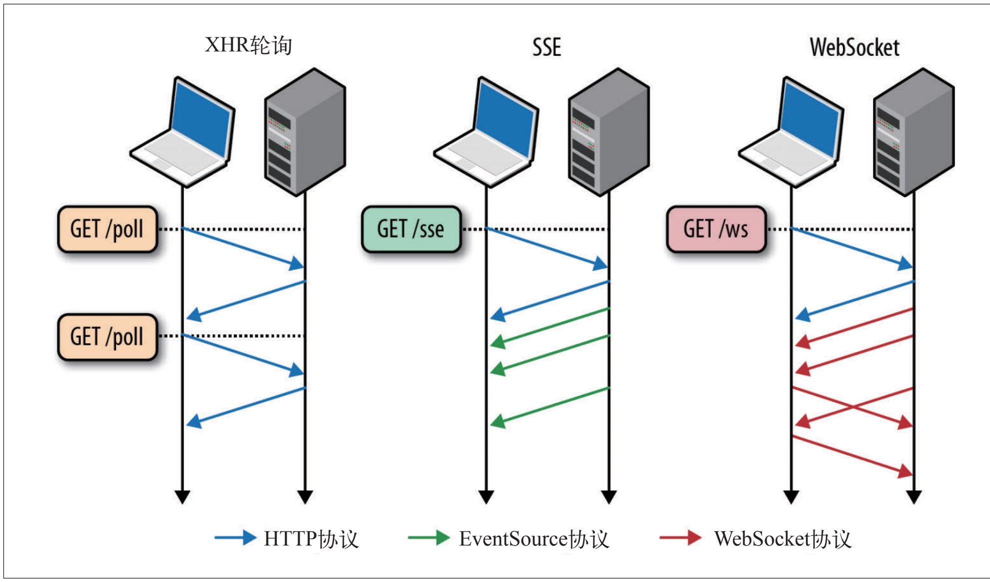


图 17-2：XHR、SSE 和 WebSocket 的通信流比较

传播与排队延迟

把传输机制从 XHR 切换为 SSE 或 WebSocket 并不会减少客户端与服务器间的往返次数！不管什么传输机制，数据包的传播延迟都一样。不过，除了传播延迟，还有一个排队延迟——消息在被发送给另一端之前必须在客户端或服务器上等待的时间。

对 XHR 轮询而言，排队延迟就是客户端轮询间隔：服务器上的消息可用之后，必须等到下一次客户端 XHR 请求才能发送（参见 15.7.1 节的“XHR 轮询的性能建模”）。相对来说，SSE 和 WebSocket 使用持久连接，这样服务器（和客户端——如果是 WebSocket）就可以在消息可用时立即发送它。

综上所述，SSE 和 WebSocket 的“低延迟交付”专指消除了消息的排队延迟。我们还没发现怎么让 WebSocket 数据包跑得比光还快！

17.3.2　消息开销

建立了 WebSocket 连接后，客户端和服务器通过 WebSocket 协议交换数据：应用消息会被拆分为一或多个帧，每个帧会添加 2~14 字节的开销。而且，由于分帧是按照自定义的二进制格式完成的，UTF-8 和二进制应用数据可以有效地通过相同的机制编码。这一点与 XHR 和 SSE 比如何呢？

- SSE 会给每个消息添加 5 字节，但仅限于 UTF-8 内容，参见 16.2 节“Event Stream 协议”。
- HTTP 1.x 请求（XHR 及其他常规请求）会携带 500~800 字节的 HTTP 元数据，加上 cookie，参见 11.5 节“度量和控制协议开销”。
- HTTP 2.0 压缩 HTTP 元数据，这样可以显著减少开销，参见 12.3.8 节“首部压缩”。事实上，如果请求都不修改首部，那么开销可以低至 8 字节！

> 记住，这里的开销数不包括 IP、TCP 和 TLS 分帧的开销，后者一共会给每个消息增加 60~100 字节，无论使用的是什么应用协议，参见 4.7.4 节“TLS 记录大小”。

17.3.3 数据效率及压缩

通过常规的 HTTP 协商，每个 XHR 请求都可以协商最优的传输编码格式（如对文本数据采用 gzip 压缩）。类似地，SSE 局限于 UTF-8 文本数据，因此事件流数据可以在整个会话期间使用 gzip 压缩。

而使用 WebSocket 时，情况要复杂一些：WebSocket 可以传输文本和二进制数据，因此压缩整个会话行不通。二进制的净荷也可能已经压缩过了！为此，WebSocket 必须实现自己的压缩机制，并针对每个消息选择应用。

好在 HyBi 工作组正在为 WebSocket 协议制定以消息为单位的压缩扩展。只是这个扩展尚未得到任何浏览器支持。因此，除非应用通过细致优化自己的二进制净荷实现自己的压缩逻辑（参见 17.1.2 节的“使用 JavaScript 解码二进制数据”），同时也针对文本消息实现自己的压缩逻辑，否则传输数据过程中一定会产生很大的字节开销！

> Chrome 和某些 WebKit 浏览器支持 WebSocket 协议压缩扩展的老版本（以帧为单位压缩），参见 17.2.2 节“WebSocket 多路复用与压缩在现实中的应用”。

17.3.4 自定义应用协议

浏览器是为 HTTP 数据传输而优化的，它理解 HTTP 协议，提供各种服务，比如认证、缓存、压缩，等等。于是，XHR 请求自然而然就继承了所有这些功能。

相对来说，流式数据处理可以让我们在客户端和服务器间自定义协议，代价是错过浏览器提供的很多服务：初次 HTTP 握手可以执行某些连接参数的协商，而一旦建

立会话，所有后续客户端与服务器间的数据流对浏览器都将是不透明的。这样来看，自定义应用协议的灵活性也有缺点，应用可能必须实现自己的逻辑来填充某些功能空白，比如缓存、状态管理、元数据交付，等等。

> 初始的 HTTP Upgrade 握手可以让服务器利用既有的 HTTP cookie 机制来验证用户。如果验证失败，服务器可以拒绝 WebSocket 升级。

利用浏览器和中间设备的缓存

使用常规 HTTP 有很多明显的优势。问自己一个简单的问题：客户端会不会因缓存接收到的数据而受益？或者中间设备如果缓存数据，是否可以优化对该数据的交付？

举个例子，WebSocket 支持二进制传输，因此应用可以流式传输任意图片而没有开销！然而，由于图片是采用自定义协议交付的，它不会被保存到浏览器或任何中间设备（如 CDN）的缓存中。结果，就可能给客户端造成不必要的下载，给来源服务器带来相当高的流量。同样的道理也适用于视频、文本等数据格式。

因此，要根据应用选择合适的传输机制！一个简单但有效的策略，就是使用 WebSocket 交付无需缓存的数据，如实时更新和应用“控制”消息，后者再触发 XHR 请求通过 HTTP 协议取得其他资源。

17.3.5　部署WebSocket基础设施

HTTP 是专为短时突发性传输设计的。于是，很多服务器、代理和其他中间设备的 HTTP 连接空闲超时设置都很激进。而这显然是我们在持久的 WebSocket 会话中所不愿意看到的。为解决这个问题，要考虑三个方面：

- 位于各自网络中的路由器、负载均衡器和代理；
- 外部网络中透明、确定的代理服务器（如 ISP 和运营商的代理）；
- 客户网络中的路由器、防火墙和代理。

我们没有权限控制客户网络的策略。事实上，某些网络甚至会完全屏蔽 WebSocket 通信，而这正是必须有备用机制的原因。类似地，我们也没有权限控制外部网络中的代理。可是，这里可以借助 TLS！通过建立一条端到端的加密信道，可以让 WebSocket 通信绕过所有中间代理。

> 使用 TLS 不会阻止中间设备超时断开空闲的 TCP 连接。可是，实践中，WebSocket 会话协商的成功率越来越高，这样也有助于增加连接超时的间隔。

最后，就是我们自己部署并管理的基础设施，需要经常关注和调优。就跟我们经常抱怨客户和外部网络一样，我们自己家的问题其实一点也不少。通信路径上的每一台负载均衡器、路由器和 Web 服务器都必须针对长时连接进行调优。

例如，Nginx 1.3.13+ 可以代理 WebSocket 通信，但默认设置了激进的 60 秒超时！为了突破这个限制，我们必须明确定义更长的超时：

```
location /websocket {
    proxy_pass http://backend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 3600; ❶
    proxy_send_timeout 3600; ❷
}
```

❶ 设置两次读操作间的超时为 60 分钟
❷ 设置两次写操作间的超时为 60 分钟

类似地，Nginx 服务器前面经常少不了要部署一台负载均衡器，比如 HAProxy。毫不奇怪，在此也要采取相同的策略，以 HAProxy 为例：

```
defaults http
  timeout connect 30s
  timeout client  30s
  timeout server  30s
  timeout tunnel  1h  ❶
```

❶ 为专用信道设置 60 分钟的不活动超时

前面例子的关键在于额外的“信道”超时。对 HAProxy 来说，`connect`、`client` 和 `server` 超时只适用于初始的 HTTP Upgrade 握手，而一旦升级完成，超时就会由信道（tunnel）值控制。

Nginx 和 HAProxy 只是我们数据中心内几百种服务器、代理和负载均衡器中的两种。我不可能在这里把所有可能的配置项都罗列出来。前面的例子只是为了说明大多数基础设施都需要自定义的配置，才能顺利处理长时会话。请记住，在实现应用的持久连接之前，首先要保证基础设施的配置正确。

> 长时连接和空闲会话会占用所有中间设备及服务器的内存和套接字资源。实际上，短超时经常被视为安全、资源管理及运维的预防措施。无论部署 WebSocket、SSE，还是 HTTP 2.0，都有赖于长时会话，都会对运维提出新的挑战。

17.4　性能检查表

部署高性能的 WebSocket 服务要求细致地调优和考量，无论在客户端还是在服务器上。可以参考下列要点。

- 使用安全 WebSocket（基于 TLS 的 WSS）实现可靠的部署。
- 密切关注腻子脚本的性能（如果使用腻子脚本）。
- 利用子协议协商确定应用协议。
- 优化二进制净荷以最小化传输数据。
- 考虑压缩 UTF-8 内容以最小化传输数据。
- 设置正确的二进制类型以接收二进制净荷。
- 监控客户端缓冲数据的量。
- 切分应用消息以避免队首阻塞。
- 合用的情况下利用其他传输机制。

最后但同样重要的是，为移动应用而优化！实时推送对于手持设备而言，反倒可能造成负面影响，因为手持设备的电池始终很宝贵。这并不代表不能在移动应用中使用 WebSocket。相反，WebSocket 其实是一个高效的传输机制，但一定要确保注意以下问题：

- 8.1 节“节约用电”；
- 8.2 节“消除周期性及无效的数据传输”；
- 8.2 节中的“内格尔及有效的服务器推送”；
- 以及 8.2 节之后的“消除不必要的长连接”。

第18章 WebRTC

Web Real-Time Communication（Web 实时通信，WebRTC）由一组标准、协议和 JavaScript API 组成，用于实现浏览器之间（端到端）的音频、视频及数据共享。WebRTC 使得实时通信变成一种标准功能，任何 Web 应用都无需借助第三方插件和专有软件，而是通过简单的 JavaScript API 即可利用。

要实现涵盖音频和视频的电话会议等完善、高品质的 RTC 应用，以及端到端的数据交换，需要浏览器具备很多新功能：音频和视频处理能力、支持新应用 API、支持好几种新网络协议。好在，浏览器把大多数复杂性抽象成了三个主要 API：

- `MediaStream`：获取音频和视频流；
- `RTCPeerConnection`：音频和视频数据通信；
- `RTCDataChannel`：任意应用数据通信。

全部功能实现起来只需十几行 JavaScript 代码，任何 Web 应用都可以立即支持全功能的电话会议，以及端到端的数据交换。这是 WebRTC 的愿景和能力！可是，上述 API 只是冰山一角：发信号、成员发现、连接协商、安全性，以及多层协议，这些还有更多组件的协同也是关键。

毫无疑问，WebRTC 的架构和协议也决定了其性能特点：连接准备延迟、协议开销，还有交付语义，只是其中一部分。事实上，与其他浏览器通信机制不同，WebRTC 通过 UDP 传输数据。不过，UDP 只是个起点，为了实现浏览器间实时通信，还有很多超出原始 UDP 之外的技术。本章我们就详细地加以讨论。

制定中的标准

WebRTC 可以为 10 多亿用户提供服务：最新的 Chrome 和 Firefox 浏览器为它们所有的用户提供 WebRTC 支持！ 话虽如此，WebRTC 标准仍然在紧张地制定中，包含浏览器 API、传输层及协议。因此，本章讨论的个别 API 和协议有可能在将来发生变化。

18.1 标准和WebRTC的发展

在浏览器中实现实时通信不可谓不雄心勃勃，毕竟，这是 Web 被发明以来最重大的一次功能增补。WebRTC 脱开我们熟悉的 C/S 通信模型，重新设计了浏览器的网络层，同时引入了全新的媒体机制，而这对于有效处理音频和视频是必需的。

正因为如此，WebRTC 架构由十余个标准组成，涵盖了应用和浏览器 API，以及很多必要的协议和数据格式：

- W3C 的 Web Real-Time Communications（WEBRTC）Working Group 负责制定浏览器 API；
- IETF 的 Real-Time Communication in Web-browsers（RTCWEB）工作组负责定义协议、数据格式、安全及其他在浏览器中实现端到端通信必需的内容。

WebRTC 并不是一个孤立的标准。虽然它的目的主要是实现浏览器间的实时通信，但设计它的时候也会考虑已有通信系统：VOIP（Voice Over IP）、各种 SIP 客户端，甚至 PSTN（Public Switched Telephone Network，公共交换电话网），等等。这些 WebRTC 标准不规定任何特定的互操作需求或 API，但它们会尽可能重用相同的概念和协议。

换句话说，WebRTC 不仅要赋予浏览器实时通信能力，而且会把浏览器的全部功能带到通信行业——2012 年产值 4.7 万亿！不必说，这个步子迈得很大，很多电信巨头、企业及创业公司都紧随其后。WebRTC 可远远不止是浏览器的一个新 API。

18.2 音频和视频引擎

要实现电话会议功能，浏览器必须访问系统硬件采集音频和视频。为此，无需第三方插件或自定义的驱动，只要使用一套简单统一的 API 即可。可是，只捕获原始音频和视频流还不够，还需要对它们分别加以处理以增强品质，保证同步，而且要适应不断变化的带宽和客户端之间的网络延迟调整输出的比特率。

接收端的处理过程相反，必须实时解码音频和视频流，并适应网络抖动和时延。总

之，采集及处理音频和视频很复杂。不过，好消息是 WebRTC 会让浏览器具备功能完备的音频和视频引擎（图 18-1），由它们替我们完成处理信号等琐碎的工作。

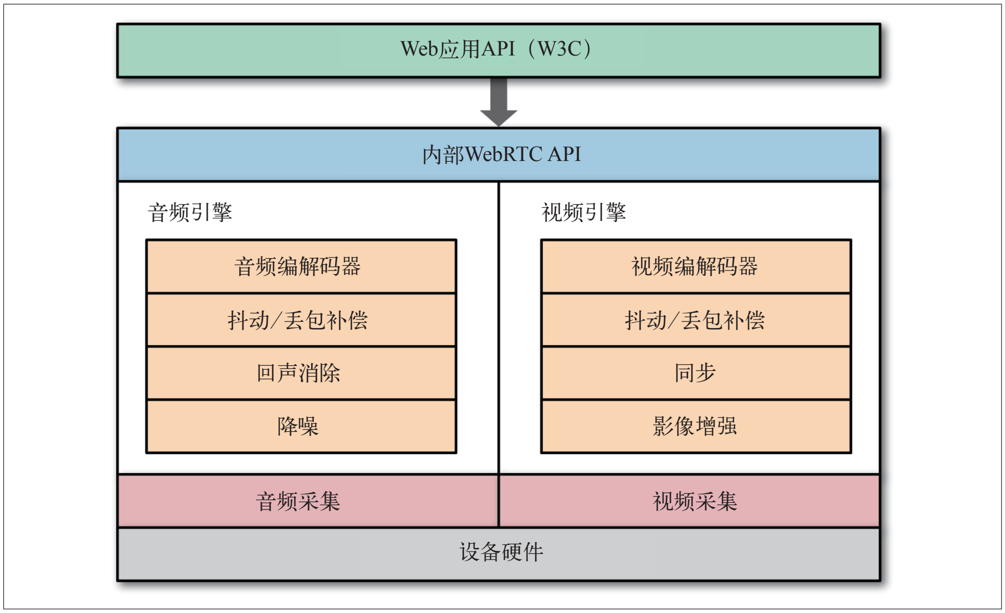


图 18-1：WebRTC 的音频和视频引擎

> 要详细介绍音频和视频引擎，轻易就能再写一本书，已经超出本书讨论范围了。更多信息，请参考：http://www.webrtc.org。

获得的音频流要经过降噪和回声消除处理，然后自动通过一种优化的窄带或宽带音频编解码器编码。关键在于，还要通过特殊的错误补偿算法消除网络抖动和丢包造成的损失，这是重点！视频引擎的流程与此类似，但着眼于影像品质，选择最优的压缩和编解码方案，应用抖动和丢包补偿，等等。

所有这一切都由浏览器负责，而且更重要的是，浏览器会动态调整其处理流程，以适应不断变化的音频和视频流及网络条件。经过浏览器这一系列处理之后，Web 应用接收到了优化的媒体流，然后可以将其输出到显示器和扬声器，发送给另外一端，或者使用 HTML5 的媒体 API 进行后期处理！

通过getUserMedia获取音频和视频

W3C 的 Media Capture and Streams 规范规定了一套新 JavaScript API，应用可以通过这套 API 从平台取得音频和视频流，并对它们进行操作和处理。其中，

`MediaStream` 对象（图 18-2）是实现这个功能的主要接口。

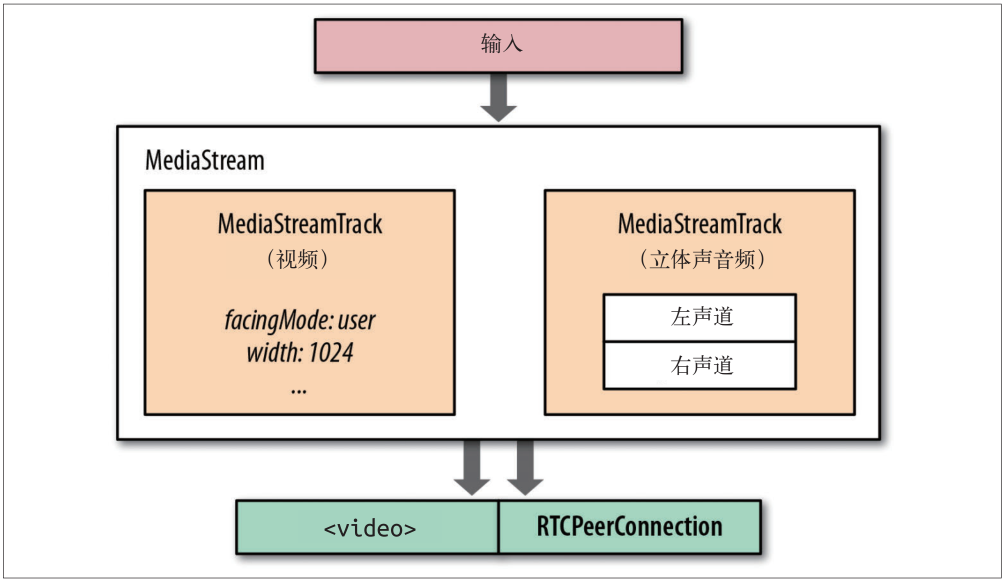


图 18-2：MediaStream 中携带着一或多个同步的 Track

- `MediaStream` 对象包含一或多个 Track（`MediaStreamTrack`）。
- `MediaStream` 中的多个 Track 相互之间是同步的。
- 输入源可以是物理设备，如麦克风、摄像头、用户硬盘或另一端远程服务器中的文件。
- `MediaStream` 的输出可以被发送到一或多个目的地：本地的视频或音频元素、后期处理的 JavaScript 代理，或者远程另一端。

`MediaStream` 对象表示一个实时的媒体流，以便应用代码从中取得数据，操作个别的 Track 和控制输出。所有的音频和视频处理，比如降噪、均衡、影像增强等都由音频和视频引擎自动完成。

不过，获得的媒体流具备什么特性，取决于输入源：麦克风只能输入音频流、某些网络摄像头能拍摄较高解析度的视频。因此，在浏览器中请求视频流时，可以通过 `getUserMedia()` API 指定一系列强制和可选的约束条件，以匹配应用的需求：

```
<video autoplay></video> ❶

<script>
  var constraints = {
    audio: true, ❷
    video: { ❸
```

```
      mandatory: {  ❹
        width: { min: 320 },
        height: { min: 180 }
      },
      optional: [  ❺
        { width: { max: 1280 }},
        { frameRate: 30 },
        { facingMode: "user" }
      ]
    }
  }

  navigator.getUserMedia(constraints, gotStream, logError);  ❻

  function gotStream(stream) { ❼
    var video = document.querySelector('video');
    video.src = window.URL.createObjectURL(stream);
  }

  function logError(error) { ... }
</script>
```

❶ HTML 的 `video` 元素，用于输出
❷ 指定请求音频 Track
❸ 指定请求视频 Track
❹ 对视频 Track 的强制约束条件
❺ 对视频 Track 的可选约束条件
❻ 从浏览器中请求音频和视频流
❼ 处理 `MediaStream` 对象的回调函数

这个例子示范了一种稍微复杂的场景：在浏览器中同时请求音频和视频 Track，同时指定了最小解析度和必须使用的摄像机类型，以及针对 720p 高清视频的一组可选约束。`getUserMedia()` API 负责获准访问用户的麦克风和摄像机，并获取符合指定要求的流——大致就这意思吧。

这里的 API 同样允许应用操作个别的 Track，复制它们，修改约束，等等。另外，取得流之后，还可以将它们提供给其他浏览器 API：

- 通过 Web Audio API 在浏览器中处理音频；
- 通过 Canvas API 采集个别视频帧并加以处理；
- 通过 CSS3 和 WebGL API 为输出的流应用各种 2D/3D 特效。

长话短说，`getUserMedia()` 就是从底层平台取得音频和视频流的简单 API。得到的媒体都经过了 WebRTC 音频和视频引擎的自动优化、编码、解码，然后可以输出到

各种目的地。这样，我们完成了实时电话会议应用的一半，剩下的就是把数据传输给另一端了！

> 要了解 Media Capture and Streams 还提供了哪些 API，请参考 W3C 官方标准：http://www.w3.org/TR/mediacapture-streams/。

音频（Opus）与视频（VP8）比特率

在通过浏览器请求音频和视频时，要注意流的大小和品质。虽然硬件有时候可以捕获到高质量的流，但 CPU 和带宽必须跟得上！当前的 WebRTC 实现使用 Opus 和 VP8 编解码器：

- Opus 编解码器用于音频，支持固定和可变的比特率编码，适合的带宽范围为 6~510 Kbit/s。这个编解码器可以无缝切换，以适应不同的带宽。
- VP8 编解码器用于视频编码，要求带宽为 100~2000+ Kbit/s，比特率取决于流的品质：
    - 720p，30 FPS：1.0~2.0 Mbit/s
    - 360p，30 FPS：0.5~1.0 Mbit/s
    - 180p，30 FPS：0.1~0.5 Mbit/s

因此，单方高清音视频流最多需要 2.5 Mbit/s 以上的带宽。如果是多方视频通话，那么考虑到要额外占用带宽、CPU 和 GPU 及内存资源，就必须降低品质。

18.3 实时网络传输

实时通信讲究的就是及时、当下。因此，处理音频和视频流的应用一定要补偿间歇性的丢包：音频和视频编解码器可以填充小的数据空白，通常对输出品质的影响也很小。类似地，应用必须实现自己的逻辑，以便因传输其他应用数据而丢包或延迟时快速恢复。及时和低延迟比可靠更重要。

> 音频和视频流还要适应人类大脑的特点。换句话说，我们的大脑非常擅长填充空白，但对延迟非常敏感。忽长忽短的几次延迟就会让人觉得“肯定是哪里出问题了”，而从中间删除一些采样数据，却很可能不会引起我们大脑的注意！

正因为及时性的要求高于可靠性，所以 UDP 协议才更适合用于传输实时数据。TCP 适合传输可靠的、有序的数据流：如果中间有丢包，TCP 就会缓冲后续所有包，等待重传，然后再严格按照次序交给应用。相对来说，UDP 则提供下列“不服务”：

- 不保证消息交付

  不确认，不重传，无超时。

- 不保证交付顺序

  不设置包序号，不重排，不会发生队首阻塞。

- 不跟踪连接状态

  不必建立连接或重启状态机。

- 不需要拥塞控制

  不内置客户端或网络反馈机制。

> 进一步讨论之前，建议大家回顾一下第 3 章，特别是 3.1 节“无协议服务”，以重温 UDP 的工作原理。

UDP 不保证数据的可靠性或者次序，只要有包到来就直接交给应用。事实上，UDP 只是对 IP 层的简单封装。

WebRTC 就使用 UDP 作为传输层协议：低延迟和及时性才是关键。因此，只要打开音频、视频，应用就会发送 UDP 包，然后就搞定了？还没那么简单。我们还是需要一些机制，穿透层层 NAT 和防火墙，为每个流协商参数，对用户数据进行加密，实现拥塞和流量控制，等等。

UDP 是浏览器实时通信的基础，但要完全达到 WebRTC 的要求，浏览器还需要位于其上的大量协议和服务的支持（图 18-3）。

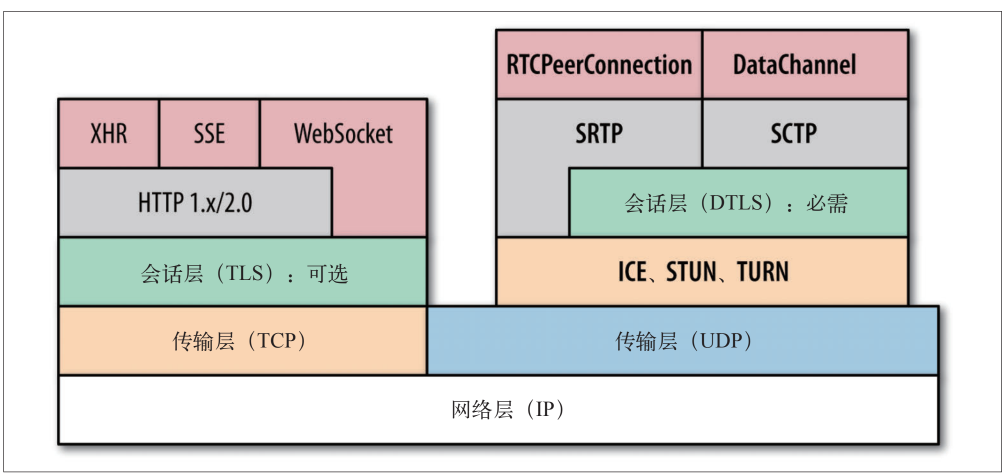


图 18-3：WebRTC 的协议分层

- ICE，即 Interactive Connectivity Establishment（RFC 5245）
  - STUN，即 Session Traversal Utilities for NAT（RFC 5389）
  - TURN，即 Traversal Using Relays around NAT（RFC 5766）
- SDP，即 Session Description Protocol（RFC 4566）
- DTLS，即 Datagram Transport Layer Security（RFC 6347）
- SCTP，即 Stream Control Transport Protocol（RFC 4960）
- SRTP，即 Secure Real-Time Transport Protocol（RFC 3711）

ICE、STUN 和 TURN 是通过 UDP 建立并维护端到端连接所必需的。DTLS 用于保障传输数据的安全，毕竟加密是 WebRTC 强制的功能。最后，SCTP 和 SRTP 属于应用层协议，用于在 UDP 之上提供不同流的多路复用、拥塞和流量控制，以及部分可靠的交付和其他服务。

没错，这么多层是有点复杂，但为了更好地理解端到端通信的性能，我们还必须先弄清楚这些层的作用。而这正是本章后面所要讨论的重点。好吧，开始。

> 不要忘了 SDP！后面会提到，SDP 是一种数据格式，用于端到端连接时协商参数。可是，SDP 的“提议与应答”（Offer/Answer）并不在系统内部，所以前面图中才没有体现它。

RTCPeerConnection API简介

尽管用于建立和维护端到端连接涉及的协议很多，但浏览器中的应用 API 相对简单。其中，`RTCPeerConnection` 接口（图 18-4）就负责维护每一个端到端连接的完整生命周期：

- `RTCPeerConnection` 管理穿越 NAT 的完整 ICE 工作流；
- `RTCPeerConnection` 发送自动（STUN）持久化信号；
- `RTCPeerConnection` 跟踪本地流；
- `RTCPeerConnection` 跟踪远程流；
- `RTCPeerConnection` 按需触发自动流协商；
- `RTCPeerConnection` 提供必要的 API，以生成连接提议，接收应答，允许我们查询连接的当前状态，等等。

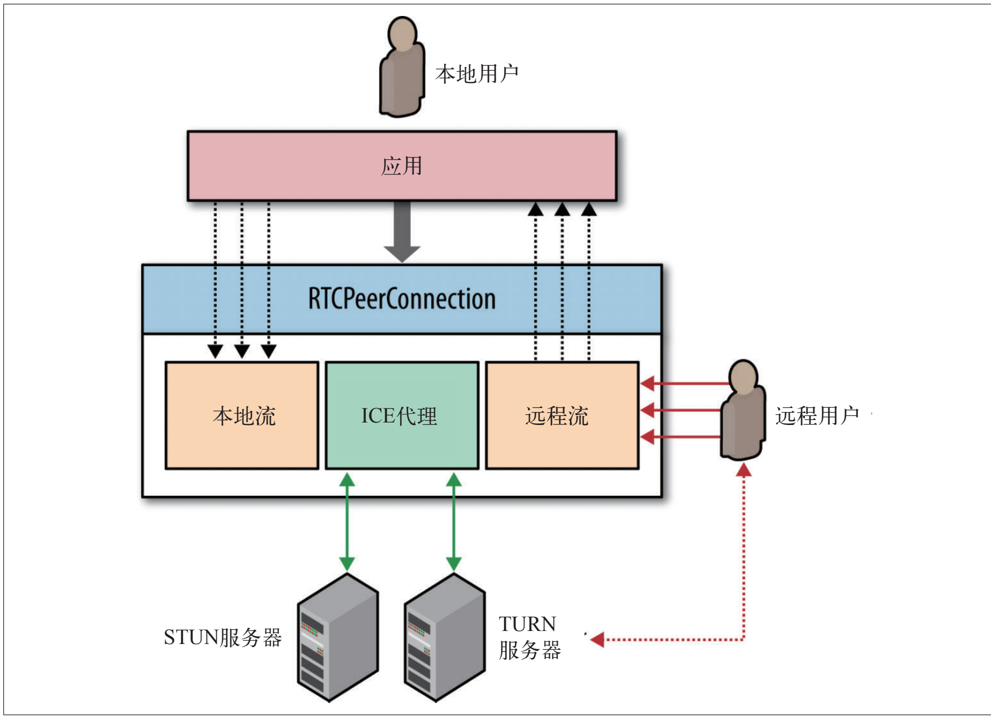


图 18-4：`RTCPeerConnection` API

简单地说，`RTCPeerConnection` 把所有连接设置、管理和状态都封装在了一个接口中。不过，在深入探讨 `RTCPeerConnection` API 的每个设置项之前，必须首先搞明白发信号和协商、提议与应答工作流，以及 ICE 穿越。下面我们就逐个介绍。

DataChannel

`DataChannel` API 用于实现端到端之间的任意应用数据交换，类似于 `WebSocket`，但却是端到端交换。而且，底层传输机制的属性也是可定制的。每个 `DataChannel` 可以经过配置提供以下特性：

- 发送消息可靠或部分可靠的交付；
- 发送消息有序或乱序交付。

不可靠的乱序交付等同于原始 UDP，即消息可能到达，也可能不到达，而且到达次序也没有保证。然而，我们可以让信道“部分可靠”，也就是设定重传的最大次数或时间限制：WebRTC 的各个层负责处理确认和超时。

信道的每一项配置都有自己的性能特点和局限，稍后还会介绍。我们继续。

18.4 建立端到端的连接

与打开 XHR、EventSource 或新 WebSocket 会话相比，初始化一个端到端的连接所做的事要多很多：前三者依赖于定义完善的 HTTP 握手机制协商连接参数，而且它们都假定客户端可以访问到目标服务器（比如，服务器具有公开的可以路由到的 IP 地址，或者客户端和服务器本身都在一个内部网中）。

相对而言，WebRTC 两端则很可能分别位于自己的私有网络中，而且中间还隔着一或多层 NAT。结果，任何一方也不能直接访问对方。为发起会话，首先必须找到两端的候选 IP 和端口（`candidate`），穿越 NAT，然后检查连接，以期找到可用路径。而即便到了这一步，也不能保证成功。

> 回顾 3.2 节“UDP 与网络地址转换器”和 3.2.2 节“NAT 穿透”，以了解 NAT 对 UDP 及端到端通信的巨大影响。

不过，尽管 NAT 穿越必须处理，但我们恐怕还是有点性急了。在打开到服务器的 HTTP 连接时，我们心里会假设服务器会监听握手。服务器当然可能拒绝，但不管怎样它会一直监听新连接。不幸的是，我们对远端没办法作这种假设：远端可能不在线或根本访问不到，亦或根本就不想与其他端建立连接。

因此，要想成功地建立端到端的连接，必须首先解决另外几个问题：

(1) 必须通知另一端我们想打开一个端到端的连接，以便它知道开始监听到来的分组；
(2) 必须找出两端之间建立连接所需的路由线路，并在两端传播这个信息；
(3) 必须交换有关媒介和数据流的必要信息，比如协议、编码，等等。

好消息是，WebRTC 为我们解决了其中一个问题：内置的 ICE 协议会执行必要的路由和连接检查。然而，发送通知（信号）和协商会话仍然要由应用负责。

18.4.1 发信号和协商会话

在检查连接或协商会话之前，必须知道能否将信息发送到另一个端，以及另一端是否愿意建立连接。为此，我们必须发出一个信号，而另一端必须返回应答（图 18-5）。但这样我们就会面临一个困境：如果另一端没有监听数据包，那我们向谁表达自己的意图呢？最低限度，我们需要一个共享的发信通道。

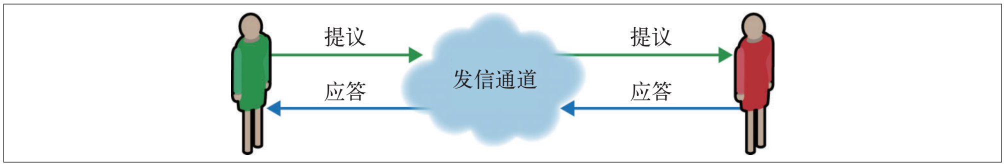


图 18-5：共享的发信通道

WebRTC 把发送信号和协议的选择交给了应用，而标准有意未给发送信号的过程提供建议或实现。为什么？这样可以让现有通信基础设施中的其他发信协议能够互操作，包括如几个协议。

- SIP（Session Initiation Protocol，*会话初始协议*）
  应用级发信协议，广泛用于通过 IP 实现的语音通话（VoIP）和视频会议。

- Jingle
  XMPP 协议的发信扩展，用于通过 IP 实现的语音通话（VoIP）和视频会议的会话控制。

- ISUP（ISDN User Part，*ISDN用户部分*）
  全球各大公共电话交换网中用于启动电话呼叫的发信协议。

> 在房间一头冲另一头喊话，也算是利用了“发信信道”。当然，你的通信对象必须能听到才行！选择什么发信媒介和协议完全是应用说了算。

WebRTC 应用可以选择已有的任何发信协议和网关（图 18-6），利用既有通信系统协商一次通话或视频会议。比如，通过 PSTN 客户端拨一通“电话”！此外，应用还可以用自定义的协议实现自己的发信服务。

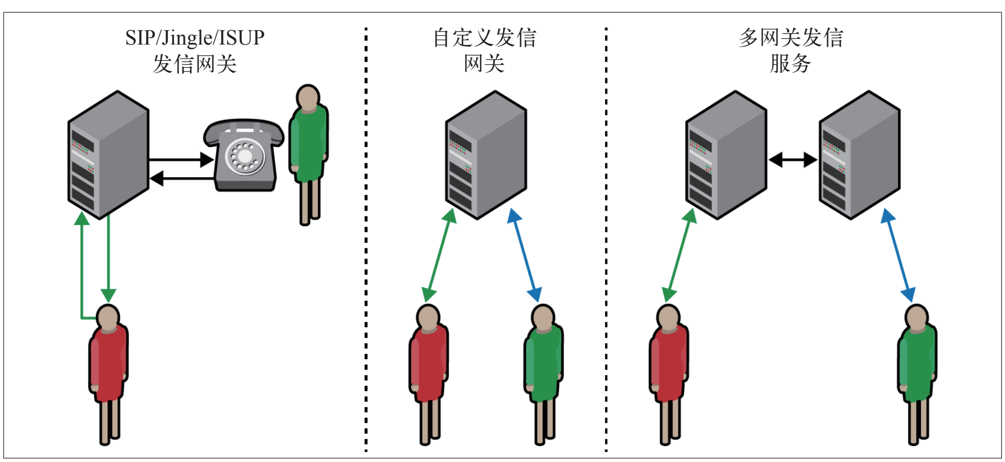


图 18-6：SIP、Jingle、ISUP 和自定义发信网关

发信服务器可以作为已有通信网络的网关，此时由网络负责将连接提议发送给目标端，然后再将应答返回给 WebRTC 客户端，以初始化信息交换。而应用也可以使用自己的自定义发信信道，可能由一或多台服务器和一个自定义通信协议构成：如果两端都连到了同一个发信服务，那这个服务就可以为它们传递消息。

> Skype 是使用自定义发信系统中的一个典型示例：音频和视频通信是端到端的，而 Skype 用户必须连接到 Skype 的发信服务器，该服务器使用自己专有的协议辅助实现端到端的连接。

选择发信服务

WebRTC 可以让我们实现端到端的通信，但每个 WebRTC 应用都需要一个发信服务器才能完成协商并建立连接。那该怎么办？

目前，可以与 WebRTC 互操作的通信网关越来越多。比如 Asterisk 就是这么一个流行、免费、开源的框架，被全球很多私人公司和大型运营商采用，以实现他们的无线电通信。对我们来说，Asterisk 有一个 WebSocket 模块，该模块支持将 SIP 作为发信协议：浏览器建立到 Asterisk 网关的 WebSocket 连接，然后两者通过交换 SIP 消息来协商会话！

此外，如果不需要与其他网络互操作，那么实现并部署自定义的发信网络对应用而言也不难。比如，一个网站可以为用户提供端到端的音频、视频和数据交换服务，而该网站会跟踪所有已经登录的用户，所以它可以对所有在线用户打开发信连接。然后，当两个用户想要建立端到端的会话时，网站的服务器就可以为两个客户端“鸿雁传书”。

选择什么发信网关要视具体情况而定，取决于应用的自身需求。不过，在考虑自定义之前，有必要先调研一下市面上有哪些开源或商业的服务可用。当然，一定要密切关注底层的发信协议，因为协议很可能严重影响发信通道的延迟和客户端与服务器的开销；参见 14.4 节“应用 API 与协议”。

18.4.2　会话描述协议（SDP）

假设应用实现了共享的发信通道，那接下来就可以执行发起 WebRTC 连接的初始步骤：

```
var signalingChannel = new SignalingChannel(); ❶
var pc = new RTCPeerConnection({}); ❷

navigator.getUserMedia({ "audio": true }, gotStream, logError);  ❸

function gotStream(stream) {
```

```
  pc.addstream(stream); ❹

  pc.createOffer(function(offer) { ❺
    pc.setLocalDescription(offer); ❻
    signalingChannel.send(offer.sdp); ❼
  });
}

function logError() { ... }
```

❶ 初始化共享的发信通道
❷ 初始化 `RTCPeerConnection` 对象
❸ 向浏览器请求音频流
❹ 通过 `RTCPeerConnection` 注册本地音频流
❺ 创建端到端连接的 SDP（提议）描述
❻ 以生成的 SDP 作为端到端连接的本地描述
❼ 通过发信通道向远端发送 SDP 提议

> 本书示例将使用不带前缀的 API，也就是 W3C 标准中定义的形式。实际上，在所有浏览器都最终实现该标准之前，你需要根据自己的浏览器调整示例代码。

WebRTC 使用 SDP（Session Description Protocol，会话描述协议）描述端到端连接的参数。SDP 不包含媒体本身的任何信息，仅用于描述“会话状况”，表现为一系列的连接属性：要交换的媒体类型（音频、视频及应用数据）、网络传输协议、使用的编解码器及其设置、带宽及其他元数据。

在前面的例子中，通过 `RTCPeerConnection` 对象注册了本地音频流之后，我们调用 `createOffer()` 生成有关会话的 SDP 描述。生成的 SDP 包含什么信息？下面我们就来看一看：

```
(……省略的内容……)
m=audio 1 RTP/SAVPF 111 ... ❶
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=candidate:1862263974 1 udp 2113937151 192.168.1.73 60834 typ host ... ❷
a=mid:audio
a=rtpmap:111 opus/48000/2 ❸
a=fmtp:111 minptime=10
(……省略的内容……)
```

❶ 带反馈的安全音频信息
❷ 媒体流的候选 IP、端口及协议

❸ Opus 编解码器及基本配置

SDP 是一个基于文本的简单协议（RFC 4568），用于描述会话属性。在前面的例子中，我们通过它描述的是采集的音频流。好消息是，WebRTC 应用不必直接处理 SDP。JSEP（ JavaScript Session Establishment Protocol ，JavaScript 会话建立协议）定义了对 RTCPeerConnection 对象几个方法的简单调用，就把 SDP 所有的内部工作全都隐藏了起来。

生成提议之后，就可以通过发信通道将它发送给远端。同样，如何编码 SDP 取决于应用：SDP 字符串可以像前面那样（作为简单的文本 blob）直接传输，也可以将它编码成任意格式后再传输。嗯，Jingle 协议提供了从 SDP 到 XMPP（XML）格式的映射。

要建立端到端的连接，两端都必须遵循一个对称的工作流（图 18-7），以交换各自音频、视频及其他数据流的 SDP 描述。

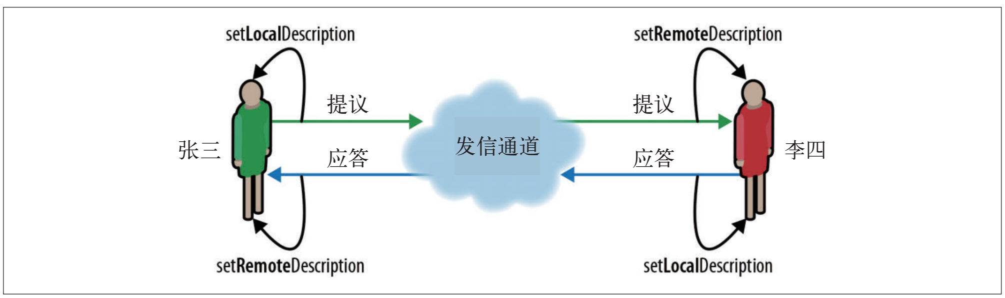


图 18-7：两端之间的 SDP 提议 / 应答交换

(1) 连接发起者（张三）通过自己的本地 RTCPeerConnection 对象注册一或多个流，创建提议，并将其设置为会话的“本地描述”。
(2) 然后，张三把生成的会话提议发送给另一端（李四）。
(3) 李四收到提议后，将张三的描述设置为会话的“远程描述”，使用他自己的 RTCPeerConnection 对象注册自己的流，生成应答的 SDP 描述，并将其设置为会话的“本地描述”。
(4) 然后，李四把生成的会话应答回传给张三。
(5) 张三接到李四的 SDP 应答后，再将这个应答设置为原始会话的“远程描述”。

这样，在通过发信通道交换 SDP 会话描述后，双方就交换了经过协商的流类型及相应设置。然后差不多马上就可以开始端到端的通信了！不过，在此之前还必须注意一个细节，那就是连接检查和 NAT 穿越。

18.4.3 交互连接建立（ICE）

按照规范，为建立端到端的连接，两端必须之间必须能收发数据包。说起来很简单，但由于端与端之间往往有很多层防火墙和NAT设备阻隔（参见3.2节“UDP与网络地址转换器”），真正实现起来并不容易。

首先，我们先看一看最简单的情况，即两端位于同一个内部网中，而且它们之间不存在防火墙或NAT设备。此时，要建立连接，两端只要查询操作系统获知IP地址（如果有多块网卡，就需要多个IP地址），将IP地址加端口号追加到生成的SDP字符串中，再把SDP转发给另一端即可。SDP交换一完成，两端就可以发起直接的端到端连接。

> 前面那个SDP示例展示的就是刚才提到的情况：其中 `a=candidate` 那一行列出的就是一个私有IP地址（192.168.x.x），供相应端发起会话（参见3.2节中的“保留的私有网络地址范围”）。

目前来看没什么问题。可是，如果其中一端或者干脆两端分别位于明显不同的私有网络中又会怎么样呢？当然还是重复前述工作流，找到并把每一端的IP地址嵌入SDP，但端到端的连接很明显无法连接成功！我们需要的是一条连接两端的公共路由线路。好在，WebRTC框架可以代替我们处理大部分复杂工作：

- 每个 `RTCPeerConnection` 连接对象都包含一个“ICE代理”；
- ICE代理负责收集IP地址和端口（`candidate`）；
- ICE代理负责执行两端的连接检查；
- ICE代理负责发送连接持久化信息。

设置好会话描述后（无论本地还是远程），本地ICE代理会自动开始发现本地端所有可能的候选IP和端口的进程：

(1) ICE代理向操作系统查询本地IP地址；
(2) 如果有配置，ICE代理会查询外部STUN服务器，以取得本地端的公共IP和端口号；
(3) 如果有配置，ICE代理会将TURN服务器追加为最后一个候选项；假如端到端的连接失败，数据将通过指定的中间设备转发。

> 如果有人问过你：“我的公共IP地址是多少？”你作了回答，那你实际上就完成了一次手工的“STUN查找”。STUN协议允许浏览器了解自己是否位于一个NAT后面，并发现自己的公共IP和端口（参见3.2.3节“STUN、TURN与ICE”）。

每发现一个新候选项（一个 IP 加一个端口号），代理就会自动通过 RTCPeerConnection 对象注册它，并通过一个回调函数（`onicecandidate`）通知应用。ICE 在完成收集工作后，也会再触发同一个回调函数，以通知应用。下面我们就来看一看 ICE 在前面的例子中所扮演的角色：

```
var ice = {"iceServers": [
              {"url": "stun:stun.l.google.com:19302"}, ❶
              {"url": "turn:user@turnserver.com", "credential": "pass"} ❷
          ]};

var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection(ice);

navigator.getUserMedia({ "audio": true }, gotStream, logError);

function gotStream(stream) {
  pc.addstream(stream);

  pc.createOffer(function(offer) {
    pc.setLocalDescription(offer); ❸
  });
}

pc.onicecandidate = function(evt) {
  if (evt.target.iceGatheringState == "complete") { ❹
      local.createOffer(function(offer) {
        console.log("Offer with ICE candidates: " + offer.sdp);
        signalingChannel.send(offer.sdp); ❺
      });
  }
}

...

// 包含 ICE 候选项的提议：
// a=candidate:1862263974 1 udp 2113937151 192.168.1.73 60834 typ host ... ❻
// a=candidate:2565840242 1 udp 1845501695 50.76.44.100 60834 typ srflx ... ❼
```

❶ STUN 服务器，配置为使用谷歌的公共测试服务器

❷ TURN 服务器，用于端到端连接失败时转发数据

❸ 应用本地会话描述：初始化 ICE 收集过程

❹ 预订 ICE 事件，监听 ICE 收集完成

❺ 生成 SDP 提议（此时包含发现的 ICE 候选项）

❻ 本地端的私有 ICE 候选项（192.168.1.73:60834）

❼ STUN 服务器返回的公有 ICE 候选项（50.76.44.100:69834）

前面的例子使用了谷歌的公共演示 STUN 服务器。可惜的是，只有 STUN 还不够（参见 3.2.3 节中的“现实中的 STUN 和 TURN”），可能还需要再提供一个 TURN，以保证那些不能直接端到端连接的用户能够建立连接（大约 8%）。

正像这个例子所展示的，ICE 代理替我们处理了大部分复杂工作：ICE 收集过程是自动触发的，STUN 查找是在后台执行的，而发现的候选项也会自动通过 RTCPeerConnection 对象注册。上述过程完成后，我们可以生成 SDP 提议，并通过发信通道发送给另一端。另一端接收到 ICE 候选项后，就可以进行第二步——建立端到端的连接了：只要 RTCPeerConnection 对象设置了远程会话描述（包含另一端的一组候选 IP 和端口号），ICE 代理就会执行连接检查（图 18-8），以确定能否抵达另一端。

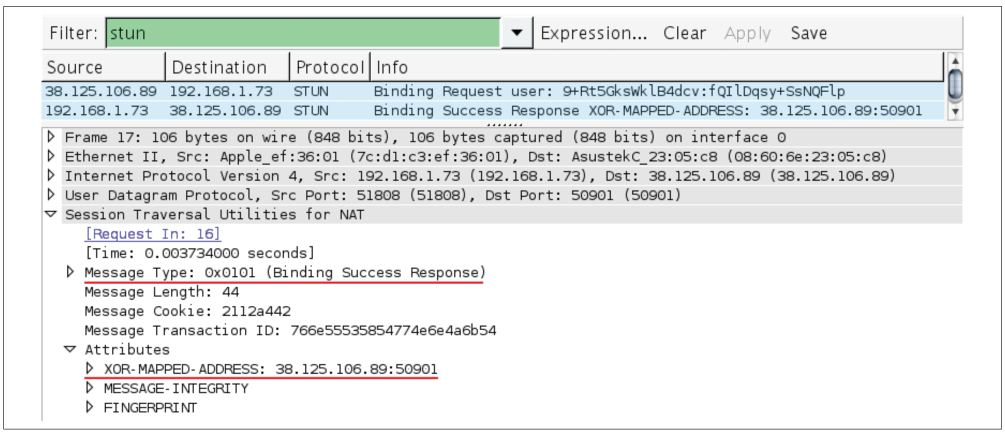


图 18-8：WireShark 中端到端 STUN 绑定请求与响应的截图

ICE 代理发送消息（STUN 绑定请求），另一端接收之后必须以一个成功的 STUN 响应确认。如果这个过程完成，那么就代表着有了一条端到端连接的路由线路！相反，如果所有候选项都绑定失败，要么将 RTCPeerConnection 标记为失败，要么回退到靠 TURN 转发服务器建立连接。

ICE 代理自动确定连接检查时候选项的次序和优先级：首先检查本地 IP 地址，然后是公共 IP，最后才检查 TURN。建立连接后，ICE 代理周期性地向另一端发送 STUN 请求，以此保证连接的持久化。

好麻烦！正如本节开头说的，初始化端到端的连接请求，比打开 XHR、EventSource 或新 WebSocket 会话要麻烦得多。好在，大部分工作都有浏览器替我们干。可是，

考虑到性能，我们也必须明白，在实际发送数据之前，可能会经历从 STUN 服务器到通信端之间的很多次往返——当然，前提是 ICE 协商成功！

18.4.4　增量提供（Trickle ICE）

ICE 收集过程决非瞬间就能完成的：取得本地 IP 地址很快，但查询 STUN 服务器需要经过到外部服务器的往返，然后还有另一次端到端的 STUN 连接检查。Trickle ICE 是对 ICE 协议的扩展，用意在于实现端与端之间的增量收集和连接检查。这个用意其实很简单：

- 两端交换没有 ICE 候选项的 SPD 提议；
- 发现 ICE 候选项之后，通过发信通道发送到另一端；
- 新候选描述一就绪，立即执行 ICE 连接检查。

简言之，就是不等到 ICE 收集过程完成，而是依靠发信通道向另一端递增地交付更新，从而加快协商。相应的 WebRTC 实现当然也很简单：

```
var ice = {"iceServers": [
              {"url": "stun:stun.l.google.com:19302"},
              {"url": "turn:user@turnserver.com", "credential": "pass"}
          ]};

var pc = new RTCPeerConnection(ice);
navigator.getUserMedia({ "audio": true }, gotStream, logError);

function gotStream(stream) {
  pc.addstream(stream);

  pc.createOffer(function(offer) {
    pc.setLocalDescription(offer);
    signalingChannel.send(offer.sdp); ❶
  });
}

pc.onicecandidate = function(evt) {
  if (evt.candidate) {
    signalingChannel.send(evt.candidate); ❷
  }
}

signalingChannel.onmessage = function(msg) {
  if (msg.candidate) {
    pc.addIceCandidate(msg.candidate); ❸
  }
}
```

❶ 发送不包含 ICE 候选项的 SDP 提议

❷ 本地 ICE 代理发现一个 ICE 候选项后就立即发送

❸ 注册远程 ICE 候选项并开始连接检查

Trickle ICE 导致发信通道的流量增加，但可以显著减少初始化端到端连接的时间。为此，所有 WebRTC 应用都应该考虑这一点：尽快发送提议，然后随着发现依次发送 ICE 候选项。

18.4.5 跟踪ICE收集和连接状态

内置的 ICE 框架负责候选项发现、连接检查、持久化，等等。如果一切顺利，那么所有这些工作对应用而言都是不可见的：我们要做的只是在初始化 `RTCPeerConnection` 对象时指定 STUN 和 TURN 服务器。可是，并非所有连接都能成功，此时隔离和解决问题就很重要了。为此，我们可以查询 ICE 代理状态并预订其通知：

```
var ice = {"iceServers": [
              {"url": "stun:stun.l.google.com:19302"},
              {"url": "turn:user@turnserver.com", "credential": "pass"}
          ]};

var pc = new RTCPeerConnection(ice);

logStatus("ICE gathering state: "  + pc.iceGatheringState); ❶
pc.onicecandidate = function(evt) { ❷
   logStatus("ICE gathering state change: " + evt.target.iceGatheringState);
}

logStatus("ICE connection state: " + pc.iceConnectionState); ❸
pc.oniceconnectionstatechange = function(evt) { ❹
  logStatus("ICE connection state change: " + evt.target.iceConnectionState);
}
```

❶ 记录当前 ICE 收集状态
❷ 预订 ICE 收集事件
❸ 记录当前 ICE 连接状态
❹ 预订 ICE 连接状态事件

顾名思义，`iceGatheringState` 属性中保存的是本地端候选项的收集状态。这个属性有 3 个可能的值。

- `new`：对象刚刚创建，还没有连网。
- `gathering`：ICE 代理正在收集本地候选项。
- `complete`：ICE 代理收集过程完成。

同样可以顾名思义，`iceConnectionState` 属性中保存着端到端的连接状态（图 18-9）。这个属性有 7 个可能的值。

- `new`：ICE 代理正在收集候选项且 / 或正在等待远程候选项的到来。
- `checking`：ICE 代理至少已经收到来自一个组件的远程候选项，而且正在检查候选项，但尚未发现连接；除了检查之外，可能仍然在收集。
- `connected`：ICE 代理已经找到一条通过所有组件的可用连接，但仍在检查其他候选项，以确定是否存在更好的连接；此时仍有可能还在收集。
- `completed`：ICE 代理已经完成收集和检查，而且发现了通过所有组件的连接。
- `failed`：ICE 代理检查完了所有候选项，但至少有一个组件的连接失败；其他一些组件的连接可能成功了。
- `disconnected`：一或多个组件的活动检查失败，相对 `failed` 更严重，在不稳定的网络上可能会间歇性触发（不需要采取什么行动）。
- `closed`：ICE 代理已经关闭，不再响应 STUN 请求。

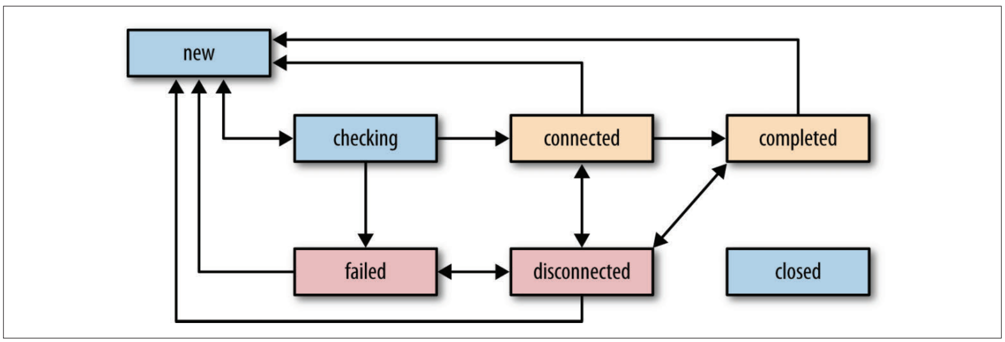


图 18-9：ICE 代理连接状态和切换

一个新的 WebRTC 会话可能需要多个流来交付音频、视频和应用数据。因此，成功的连接应该针对所有请求的流都能建立连接。而且，由于端到端连接本身并不可靠，也不能保证连接建立后会一直可用：连接可能会周期性地在连接和断开状态之间来回切换，而 ICE 代理在重新建立连接时，也会尝试选择最佳路径。

> ICE 代理最重要的目标，就是识别端到端之间的可行路径。可是，ICE 代理不会就此止步。即便是连接已经建立，ICE 代理也可能周期性地尝试其他候选项，以确定其他路径的性能是否更好。

使用谷歌 Chrome 浏览器检测 WebRTC 连接状态

谷歌 Chrome 为检查任何 WebRTC 连接的工作流和状态提供了一个简单且非常有用的工具。打开新标签页，加载 chrome://webrtc-internals。在这个标签页里，可以检查所有打开的端到端连接（图 18-10），查看交换的 SDP 描述，以及更多。

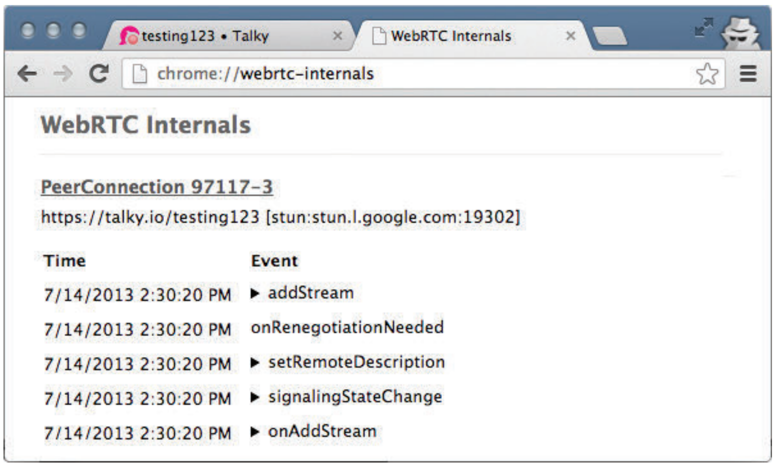


图 18-10：通过 chrome://webrtc-internals 查看 WebRTC 连接状态

Chrome 还会报告每个流的一些统计数据，比如可用带宽、延迟、编码视频和音频的比特率，等等。即便你不开发 WebRTC 应用，只为了解 WebRTC 的工作原理，你也可以与朋友或在多个浏览器窗口间开一个 WebRTC 会话，然后打开 chrome://webrtc-internals，一目了然！确实是非常难得的一个好帮手。

18.4.6 完整的示例

前面介绍了很多基础知识，包括发信号、提议 – 应答工作流、通过 SDP 协商会话参数，也深入探讨了 ICE 协议在建立端到端连接时的内部工作过程。最后，我们知道了通过 WebRTC 初始化端到端连接的所有必要细节。

1. 初始化WebRTC连接

了解了所有基本知识之后，下面我们就来看一个初始化 WebRTC 连接的完整的例子：

```
<video id="local_video" autoplay></video> ❶
<video id="remote_video" autoplay></video> ❷

<script>
```

```
    var ice = {"iceServers": [
                  {"url": "stun:stunserver.com:12345"},
                  {"url": "turn:user@turnserver.com", "credential": "pass"}
              ]};

    var signalingChannel = new SignalingChannel(); ❸
    var pc = new RTCPeerConnection(ice); ❹

    navigator.getUserMedia({ "audio": true, "video": true }, gotStream, logError); ❺

    function gotStream(evt) {
      pc.addstream(evt.stream); ❻

      var local_video = document.getElementById('local_video');
      local_video.src = window.URL.createObjectURL(evt.stream); ❼

      pc.createOffer(function(offer) { ❽
        pc.setLocalDescription(offer);
        signalingChannel.send(offer.sdp);
      });
    }

    pc.onicecandidate = function(evt) { ❾
      if (evt.candidate) {
        signalingChannel.send(evt.candidate);
      }
    }

    signalingChannel.onmessage = function(msg) { ❿
      if (msg.candidate) {
        pc.addIceCandidate(msg.candidate);
      }
    }

    pc.onaddstream = function (evt) { ⓫
      var remote_video = document.getElementById('remote_video');
      remote_video.src = window.URL.createObjectURL(evt.stream);
    }

    function logError() { ... }
  </script>
```

❶ 输出本地流的视频元素

❷ 输出远程流的视频元素

❸ 初始共享的发信通道

❹ 初始端连接对象

❺ 取得本地音频和视频流

❻ 通过端连接注册本地 MediaStream

❼ 把本地视频流输出到视频元素（本地视图）

❽ 生成描述端连接的 SPD 提议并发送到对端
❾ 通过发信通道向对端增量发送 ICE 候选项
❿ 注册远程 ICE 候选项以开始连接检查
⓫ 把远程视频流输出到视频元素（远程视图）

看第一遍可能会让人觉得整个过程有点难理解，不要灰心。既然我们已经理解了每一步的工作过程，整个过程也不如此：初始端连接和发信通道，取得并注册媒体流，发送提议，递增提交 ICE 候选项，最后再输出取得的媒体流。如果实现再完整一些，还应该注册更多回调函数，以跟踪 ICE 收集和连接状态，从而为用户提供更多反馈。

> 建立连接后，应用仍然可以通过 RTCPeerConnection 对象添加或移除流。每次添加或移除后，都会自动触发新的一次 SDP 协商，重复一遍初始化过程。

2. 响应WebRTC连接

响应新 WebRTC 连接请求的过程十分类似，主要区别是在发信通道交付 SDP 提议之后才会开始执行大部分逻辑。下面我们就实际地看一看：

```
<video id="local_video" autoplay></video>
<video id="remote_video" autoplay></video>

<script>
  var signalingChannel = new SignalingChannel();

  var pc = null;
  var ice = {"iceServers": [
              {"url": "stun:stunserver.com:12345"},
              {"url": "turn:user@turnserver.com", "credential": "pass"}
            ]};

  signalingChannel.onmessage = function(msg) {
    if (msg.offer) { ❶
      pc = new RTCPeerConnection(ice);
      pc.setRemoteDescription(msg.offer);

      navigator.getUserMedia({ "audio": true, "video": true },
      gotStream, logError);

    } else if (msg.candidate) { ❷
      pc.addIceCandidate(msg.candidate);
    }
  }

  function gotStream(evt) {
```

```
      pc.addstream(evt.stream);

      var local_video = document.getElementById('local_video');
      local_video.src = window.URL.createObjectURL(evt.stream);

      pc.createAnswer(function(answer) { ❸
        pc.setLocalDescription(answer);
        signalingChannel.send(answer.sdp);
      });
    }

    pc.onicecandidate = function(evt) {
      if (evt.candidate) {
        signalingChannel.send(evt.candidate);
      }
    }

    pc.onaddstream = function (evt) {
      var remote_video = document.getElementById('remote_video');
      remote_video.src = window.URL.createObjectURL(evt.stream);
    }

    function logError() { ... }
  </script>
```

❶ 监听并处理通过发信通道交付的远程提议
❷ 注册远程 ICE 候选项以开始连接检查
❸ 生成描述端连接的 SDP 应答并发送到对端

毫不奇怪，连代码看起来都十分相似。除了基于由共享的发信通道交付的提议消息初始化端连接的工作流，唯一的重要区别就在于前面的代码生成的是一个 SDP 应答（通过 `createAnswer`），而非提议对象。除此之外，整个过程是对称的：初始化端连接，取得并注册媒体流，发送应答，增量提交 ICE 候选项，最后输出取得的媒体流。

既然是这样，我们只要复制这些代码，再加上发信通道的实现，就有了一个在浏览器中实时的、端到端的、支持视频和音频的视频会议应用。想一想，只需 100 行 JavaScript 代码，不错啦！

通过 SimpleWebRTC 初始化 WebRTC 会话

实践中，前面的代码还可以进一步简化。我们这个例子是以手工方式把所有工作串接起来的，但实际上，没有理由不把其中大多数代码封装到一个库里。就举一个使用 `simpleWebRTC` 库的例子吧：

```
<script src="http://simplewebrtc.com/latest.js"></script>

<div id="local_video"></div>
<div id="remote_video"></div>

<script>
  var webrtc = new WebRTC({
    localVideoEl: "local_video",
    remoteVideosEl: "remote_video",
    autoRequestMedia: true
  });

  webrtc.on("readyToCall", function () {
      webrtc.joinRoom("your awesome room name");
  });
</script>
```

以上这几行 JavaScript 代码能实现与前面例子一样的视频会议功能。可是，这里也没有什么神秘的，只不过 simpleWebRTC 替我们做了一些决定而已。在这几行代码的背后，simpleWebRTC 使用一个用于穿透 NAT 的公共 STUN 服务器初始化了 RTCPeerConnection，使用 getUserMedia 请求音频和视频流，并初始化了连接到它自己发信服务器的 WebSocket 连接。而应用要做的，只是指定一个“房间名”，那是所有想建立端到端连接的节点都必须认可的。

其他细节请大家参考 simpleWebRTC 的文档（http://simplewebrtc.com/）。值得一提的是，这个项目还提供了一个开源的发信服务器，你可以利用它，也可以在实现自己的发信服务器时参考它。

18.5 交付媒体和应用数据

建立端到端的连接需要花很多工夫。可是，即便两端完成了提议 – 应答工作流，每端也完成了 NAT 穿越和 STUN 连接检查，在 WebRTC 协议栈中仍然只走了一半的路。此时，两端相互都打开了原始的 UDP 连接，可以传输基本的数据报，但我们知道仅仅这样是不行的（参见 3.3 节“针对 UDP 的优化建议”）。

没有流量控制、拥塞控制、错误校验，以及一些预测带宽和延迟的机制，很容易把网络搞得一团糟，而这又会进一步降低两端及周边节点的性能。另外，UDP 以明文传输数据，而 WebRTC 要求对所有通信加密！为解决这个问题，WebRTC 又在 UDP 之上增加了几层协议：

- 数据报传输层安全（DTLS，Datagram Transport Layer Security），用于加密传输应用数据时针对要传输的媒体数据协商密钥；
- 安全实时传输（SRTP，Secure Real-Time Transport），用于传输音频和视频流；

• 流控制传输协议（SCTP，Stream Control Transport Protocol）用于传输应用数据。

18.5.1　通过DTLS实现安全通信

WebRTC 规范要求所有传输的数据（音频、视频和自定义应用数据）都必须加密。当然啦，如果不是因为有赖于 TCP 的有序交付，TLS 是最合适的。既然是 UDP 连接，所以 WebRTC 就使用 DTLS，它能提供与 TLS 相同的安全保护。

DTLS 在设计上有意与 TLS 保持一致，事实上，DTLS 本质上就是 TLS，只是为了兼容 UDP 的数据报传输而做了一些微小的修改。特别地，DTLS 解决了下列问题：

(1) TLS 要求可靠的有序的适合分段的握手记录以协商信道；
(2) 如果在混合多个分组的基础上对记录分段，就不能保证 TLS 的完整性校验；
(3) 如果记录的顺序不对，也不能保证 TLS 的完整性校验。

> 要了解握手过程和记录协议结构的详细信息，请参考 4.2 节“TLS 握手”和 4.6 节“TLS 记录协议”。

要解决 TLS 握手顺序的问题并不简单：每条记录都有特定用途，而且必须按照握手算法规定的次序发送，而有些记录轻易就会包含多个分组。结果，DTLS 就针对握手顺序实现了一个“迷你 TCP”（图 18-11）。

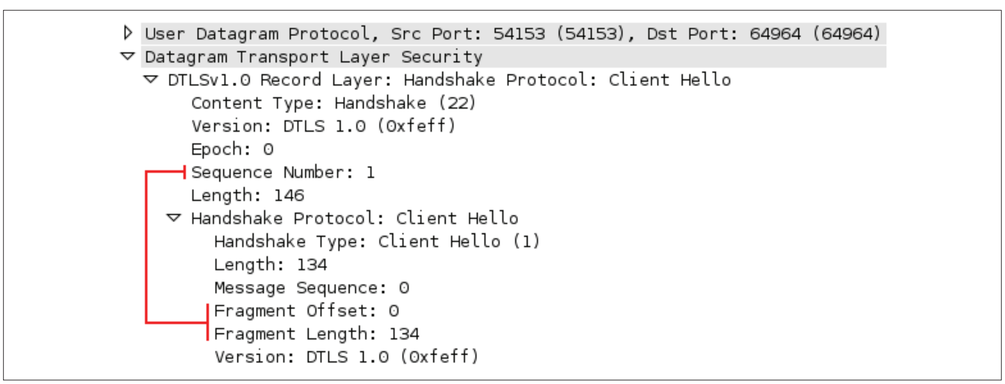


图 18-11：DTLS 握手记录中的序号和分段偏移字段

DTLS 对 TLS 记录协议的扩展，就是为每条握手记录明确添加了分段偏移字段和序号。这样就满足了有序交付的条件，也能让大记录可以被分段成多个分组并在另一端再进行组装。DTLS 握手记录严格按照 TLS 协议规定的顺序传输，顺序不对就报错。最后，DTLS 还要处理丢包问题：两端都使用计时器，如果预定时间内没有收到应答，就重传握手记录。

记录序号、偏移值和重传计时器让 DTLS 在 UDP 之上实现了握手（图 18-12）。为保证过程完整，两端都要生成自已签名的证书，然后按照常规的 TLS 握手协议走。

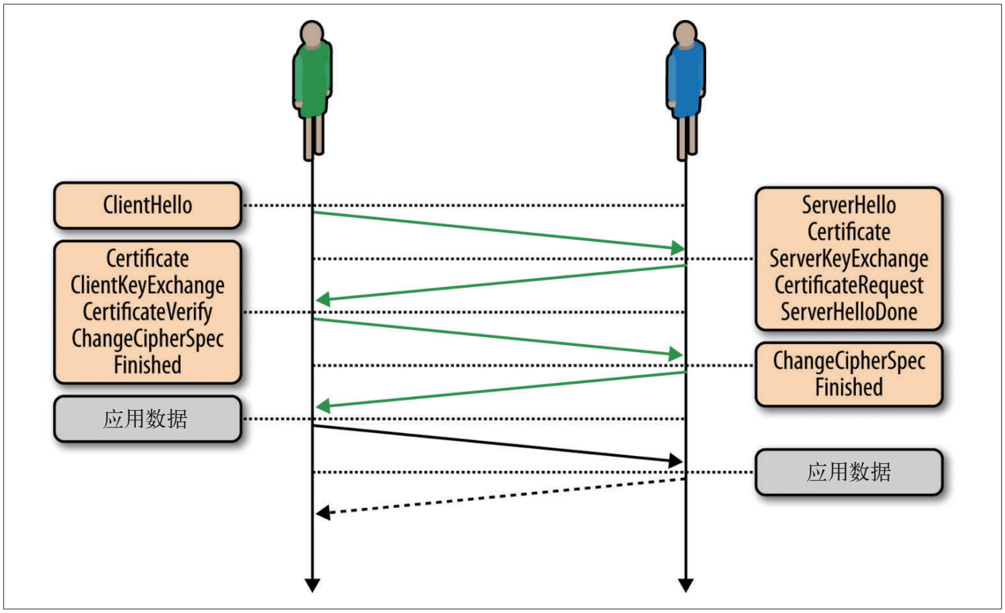


图 18-12：通过 DTLS 实现端到端的握手

> 完整的 DTLS 握手需要两次往返，这一点必须牢记。换句话说，建立端到端的连接会产生额外延迟。

WebRTC 客户端自动为每一端生成自已签名的证书。因此，也就没有证书链需要验证。DTLS 保证了加密和完整性，但把身份验证工作留给了应用；参见 4.1 节“加密、身份验证与完整性”。最后，在满足握手要求的基础上，DTLS 为处理常规记录可能出现的分段和乱序问题，又增加了两条重要的规则：

- DTLS 记录必须刚好放到一个网络分组中；
- 必须有一个分组密码用于加密记录数据。

常规 TLS 记录最大可以达到 16 KB。TCP 可以处理分段和组装，但 UDP 不提供这些服务。结果，为适应 UDP 协议的乱序发送，也为了最大程度保持其语义，每个携带应用数据的 DTLS 记录都必须放到一个 UDP 分组中。类似地，由于它们潜在依赖记录数据的有序发送，因此也不允许使用流密码。

> **身份与验证**
>
> WebRTC 两端之间的 DTLS 握手有赖于自签名证书。而这样的证书不能用于验证身份，因为没有要验证的信任链（4.4 节“信任链与证书颁发机构”）。必要的情况下，WebRTC 应用必须自己对参与各端进行认证和身份验证：
>
> - Web 应用可以利用之前用于建立 WebRTC 会话的身份验证系统（比如通过登录来验证用户）；
> - 此外，每一端也可以在生成 SDP 提议 / 应答时指定各自的“身份颁发机构”，等对端接收到 SDP 消息后，可以联系指定的身份颁发机构验证收到的证书。
>
> 后一种“身份颁发机构”机制是 W3C WebRTC 工作组目前还在讨论和制定的一种机制。要了解最新进展，可以阅读相应的规范和邮件列表。

18.5.2 通过SRTP和SRTCP交付媒体

WebRTC 以完全托管的形式提供媒体获取和交付服务：从摄像头到网络，再从网络到屏幕。WebRTC 应用指定获取流的媒体约束，然后通过 `RTCPeerConnection` 对象注册它们（图 18-13）。从此以后，就都是浏览器提供的 WebRTC 媒体和网络引擎的事了：编码优化、处理丢包、网络抖动、错误恢复、流量、控制，等等。

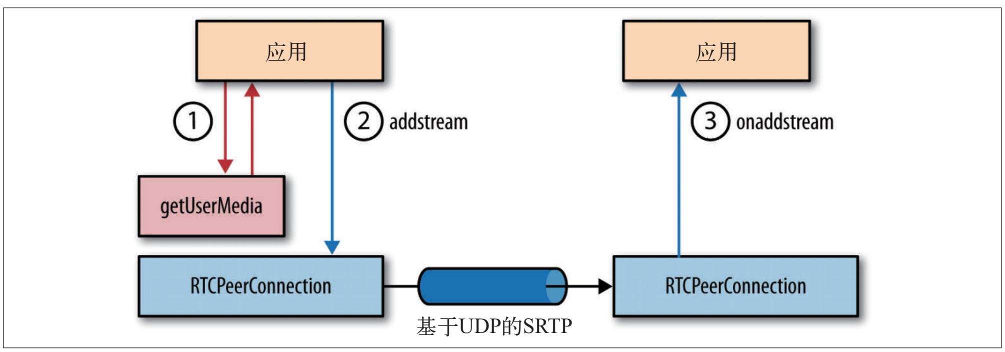


图 18-13：通过 SRTP 和 SRTCP 交付音频和视频

这种架构设计的用意很简单，就是应用除了在一开始指定媒体流的约束外（如 720p 还是 360p 视频），无法直接控制媒体如何优化以及如何交付给另一端。这样的设计是有意为之的，毕竟，通过带宽波动、分组延迟的不可靠传输机制交付高品质、实时的音频和视频，不是一件容易的事。浏览器可以替我们做这些事。

- 不用关心提供的媒体流的品质和大小，网络组件会实现自己的流和拥塞控制算法，让每个连接开始时的比特率保持较低（<500 Kbit/s），然后再根据带宽调整流的品质；

- 在连接的生命周期中，浏览器中的媒体和网络引擎会动态调整流的品质，以适应不断变化的网络环境，比如带宽波动、丢包、网络抖动，等等。换句话说，WebRTC 会生成自适应的比特流（参见 6.4.2 节中的“自适应比特流”）。

WebRTC 不能保证应用提供的高清视频流会以最高品质交付，因为端到端之间的带宽可能不够用，而且丢包现象很常见。换句话说，引擎会适应网络条件交付提供的流。

> 虽然实际交付的音频或视频流的品质可能比应用最初取得的流的品质低，但反过来则不会：WebRTC 不会提升流的品质。如果应用设置了 360p 的视频约束，那么也就设定了交付视频时占用带宽的上限。

WebRTC 是怎么优化和调整媒体流的品质的呢？ WebRTC 实际上并不是通过 IP 网络实时交付音频和视频的第一个应用。WebRTC 只是重用了 VoIP 电话使用的传输协议、通信网关和各种商业或开源的通信服务：

- 安全实时传输协议（SRTP，Secure Real-time Transport Protocol）
  通过 IP 网络交付音频和视频等实时数据的标准安全格式。

- 安全实时控制传输协议（SRTCP，Secure Real-time Control Transport Protocol）
  通过 SRTP 流交付发送和接收方统计及控制信息的安全控制协议。

> 实时传输协议（RTP，Real-Time Transport Protocol）由 RFC 3550 定义。然而，WebRTC 要求传输中的所有数据都必须加密。因此，WebRTC 使用的是 RTP 的“安全版”（RFC 3711）。SRTP 和 SRTCP 前面那个 S，就是 Secure（安全）。

SRTP 为通过 IP 网络交付音频和视频定义了标准的分组格式（图 18-14）。SRTP 本身并不对传输数据的及时性、可靠性或数据恢复提供任何保证机制，它只负责把数字化的音频采样和视频帧用一些元数据封装起来，以辅助接收方处理这些流。

位	+0..7	+8..15	+16..23	+24..31
0	V | P | X | CC	M | 净荷类型	序号	
32	时间戳			
64	同步源（SSRC，Synchronization source）标识符			
+32	参与源（CSRC，Contributing source）标识符（可选）			
+32	RTP扩展（可选）			
+N	加密的RTP净荷			
...	SRTP MKI（可选）+认证标签（可选）			

图 18-14：SRTP 首部（12 字节 + 净荷及可选字段）

- 每个 SRTP 分组都包含一个自动递增的序号，以便接收端检测和发现媒体数据是否乱序；
- 每个 SRTP 分组都包含一个时间戳，表示媒体净荷第一字节的采样时间，用于多个媒体流（如音频和视频）的同步；
- 每个 SRTP 分组都包含一个 SSRC 标识符，这是个别媒体流中每个分组的唯一流 ID；
- 每个 SRTP 分组可以包含其他可选的元数据；
- 每个 SRTP 分组都包含加密的媒体净荷，以及（可选的）认证标签，后者用于验证分组的完整性。

SRTP 分组中包含了媒体引擎实时回放流必需的所有信息。而控制每个 SRTP 分组交付则是 SRTCP 协议的责任，SRTCP 针对每个媒体流实现了独立的外部反馈渠道。

SRTCP 会跟踪发送及丢失字节和分组的数量，跟踪每个 SRTP 分组的序号、交错到达抖动，以及其他 SRTP 统计信息。然后，两端定时交换这些数据，以便调整每个流的发送速率、编码品质和其他参数。

简单地说，SRTP 和 SRTCP 直接在 UDP 之上运行，共同完成对应用提供的音频和视频流的实时适配和优化。WebRTC 应用不会接触内部的 SRTP 或 SRTCP 协议：如果你要构建自定义的 WebRTC 客户端，那得直接操作这两个协议，否则浏览器会替你搭建好所有必要的基础设施。

> 真想看看 WebRTC 会话的 SRTCP 统计信息？ Chrome 就可以让你看到延迟、比特率和带宽等情况，详情请见 18.4.5 节中的“使用谷歌 Chrome 浏览器检测 WebRTC 连接状态”。

适应 WebRTC 的 SRTP 和 SRTCP

我们对 SRTP 和 SRTCP 的介绍简单但重点突出，不过对实现来说，为了让这两个协议达到 WebRTC 的要求，还需要考虑另外一些细节。

- SRTP 和 SRTCP 都会加密应用净荷数据（WebRTC 的要求），但它们都没有提供协商密钥的机制！这就是为什么必须先进行 DTLS 握手的原因：DTLS 握手会为两端确定一个共享密钥，随后的 SRTP 和 SRTCP 可以使用这个密钥。
- SRTP 和 SRTCP 都要求对不同的流分配不同的端口，而这对于 NAT 或防火墙后面的客户端当然就是一个问题。为解决这个问题，WebRTC 使用了另一个多路复用扩展，以便向同一个目标端口交付多个流（以及相应的控制信道）。
- IETF 还制定了一个新的拥塞控制算法，该算法利用 SRTCP 的反馈对 WebRTC 应用生成的音频和视频流进行优化。

总之，并不是仅仅把数字化的音频和视频数据转换成 UDP 分组那么简单。所幸的是，WebRTC 的媒体和网络引擎替我们承担了所有复杂的工作。而适配和改进 SRTP 和 SRTCP 性能仍然是一个有待研究的领域，包括标准和实现层面。

18.5.3 通过SCTP交付应用数据

除了传输音频和视频数据，WebRTC 还支持通过 DataChannel API 在端到端之间传输任意应用数据。上一节介绍的 SRTP 协议是专门为传输媒体数据而设计的，不适合传输应用数据。因此，DataChannel 就依赖于 SCTP（Stream Control Transmission Protocol，流控制传输协议），而 SCTP 在两端之间建立的 DTLS 信道之上运行（图 18-3）。

别急，在探讨 SCTP 协议之前，我们先了解一下 WebRTC 对 RTCDataChannel 接口及其传输协议有哪些要求。

- 传输层必须支持多个独立信道的复用：
  - 每个信道必须支持有序或乱序交付；
  - 每个信道必须支持可靠或不可靠交付；
  - 每个信道可以支持应用定义的优先级。
- 传输层必须提供一个面向消息的 API：
  - 每条应用消息都可能在传输层被分段和组装。
- 传输层必须实现流量和拥塞控制机制。
- 传输层必须保证数据的机密性和完整性。

好消息是，DTLS 可以满足最后一条要求：所有应用数据在记录的净荷中都会得到加密，因此机密性和完整性就落实了。不过，其他要求可没那么容易满足。UDP 提供的是不可靠、乱序的数据报交付，而除此之外我们还需要 TCP 似的可靠交付、信道复用、优先级支持、消息分段，等等。这才有了 SCTP。

表18-1：TCP、UDP与SCTP比较

	TCP	UDP	SCTP
可靠性	可靠	不可靠	可配置
交付次序	有序	乱序	可配置
传输方式	面向字节	面向消息	面向消息
流量控制	支持	不支持	支持
拥塞控制	支持	不支持	支持

SCTP 是一个传输层协议，直接在 IP 协议上运行，这一点跟 TCP 和 UDP 类似。不过在 WebRTC 这里，SCTP 是在一个安全的 DTLS 信道中运行，而这个信道又运行在 UDP 之上。

SCTP 同时具备 TCP 和 UDP 中最好的功能：面向消息的 API、可配置的可靠性及交付语义，而且内置流量和拥塞控制机制。我们不打算全面剖析 SCTP 协议，但可以简单了解一下它的某些概念和术语。

- 关联（association）
  连接的同义词。

- 流（stream）
  单向消息传输信道，应用消息在其中有序交付；通过配置这个信道，可以实现乱序交付。

- 消息（message）
  提交给这个协议的应用数据。

- 块（chunk）
  SCTP 分组中的最小通信单位。

两个端点之间的一个 SCTP 关联可以容纳多个独立的流，每个流都可以独立传输应用消息。而每个应用消息又可以被分割成一或多个块，这些块被封装在 SCTP 分组（图 18-15）中交付，等到了另一端再组装起来。

位	+0..7	+8..15	+16..23	+24..31
0	来源端口		目标端口	
32	验证标签			
64	校验和			
96	类型（0）	保留 U B E	长度	
128	传输序号（TSN）			
160	流标识符		流序号	
192	净荷协议标识符			
224	净荷			

图 18-15：SCTP 首部和数据块

这些概念和解释听起来是不是很熟悉？必须的呀！名词不同，但核心概念与 HTTP 2.0 分帧层中那些概念是一样的；参见 12.3.2 节“流、消息和帧”。这里的不同在

于，SCTP 是在“较低层”实现同样的功能，从而可以支持任意应用数据的有效传输及多路复用。

SCTP 分组由公共首部及一或多个控制字段或数据块组成。首部 12 字节，用于标识来源和目标端口、针对当前 SCTP 关联随机生成的验证标签，以及整个分组的校验和。首部后面是一或多个控制字段或数据块，图 18-15 中所示的数据报只包含一个数据块。

- 所有数据块的类型都是 0×0。
- U（unordered）位表示数据块是不是乱序数据块。
- B 和 E 位用于表示分成多个数据块的消息的起止位置：B=1，E=0 表示消息的开始位置；B=0，E=0 表示中间位置；B=0，E=1 表示末尾位置；B=1，E=1 表示没有分段的消息。
- 长度表示数据块的大小，包括首部(比如，16 字节的块首部，加上净荷数据的大小)。
- TSN（传输序号）是一个 32 位数，SCTP 内部使用它确认收到分组及检测重复的分组。
- 流标识符表示当前数据块所属的流。
- 流序号是一个自动递增的消息编号，表示关联的流；分段消息的流序号相同。
- PPID（净荷协议标识符）是一个自定义字段，由应用填写，用于沟通与传输的块相关的其他元数据。

> `DataChannel` 使用 SCTP 首部的 PPID 字段标记传输的数据类型：0×51 表示 UTF-8，0×52 表示二进制应用净荷。

一下子说了这么多，可能不太好记。再重复一遍，这次我们在前面提到的 WebRTC 和 `DataChannel` API 要求的背景下讨论。

- SCTP 首部包含一些冗余字段：SCTP 信道建立在 UDP 之上，后者已经指定了来源和目标端口（参见图 3-2）。
- SCTP 利用首部中的 B、E 和 TSN 字段辅助处理消息分段：可以标识每个块的位置（开始、中间、末尾），TSN 值用于表示中间块的次序。
- SCTP 支持流的多路复用：每个流都有一个唯一的流标识符，用于关联当前数据块与活动的那个流。
- SCTP 为每条应用消息指定一个独特的序号，利用该序号可以实现有序交付的语义。可选地，如果设置了 U 位（乱序交付），SCTP 可以继续使用这个序号来处理消息分段，但消息可以乱序交付。

> 总之，SCTP 会给每个数据块增加 28 字节开销：12 字节的公共首部和 16 字节的数据块首部，然后才是应用净荷。

SCTP 怎么协商关联的初始参数呢？每个 SCTP 连接都需要经历与 TCP 类似的握手过程！类似地，SCTP 也实现了 TCP 友好的流量和拥塞控制机制：两个协议使用相同的初始拥塞窗口大小，实现了与拥塞预防阶段类似的增减拥塞窗口的逻辑。

> 要回顾一下 TCP 握手延迟、慢启动和流量控制，请参考第 2 章。WebRTC 中使用的 SCTP 握手以及拥塞和流量控制算法不一样，但目的相同，而且在性能方面的开销也类似。

到现在为止，差不多已经能够满足 WebRTC 的所有要求了。然而可惜的是，就算有了这么多功能，还是欠缺几个关键的特性。

(1) 基础的 SCTP 标准（RFC 4960）提供了乱序交付消息的机制，但不支持通过配置实现可靠性。为解决这个问题，WebRTC 客户端必须另外求助“Partial Reliability Extension”（RFC 3758），也就是对 SCTP 的扩展，为的是支持发信端实现自定义的交付保证，而这个可靠性是 DataChannel 的关键特性。
(2) SCTP 不支持对流的优先级安排，协议没有规定用于保存优先级的字段。因此，这个功能必须由更高层协议来实现。

简言之，SCTP 与 TCP 提供类似服务，因为它运行于 UDP 之上，而且由 WebRTC 客户端实现，所以它的 API 必须更强大：有序和乱序交付、部分可靠性、面向消息的 API，等等。同时，SCTP 也有握手延迟、慢启动以及流量和拥塞控制，这些都是在考量 `DataChannel` API 性能时不能忽略的关键因素。

“裸 SCTP”的挑战

使用面向消息的 API 让 SCTP 避免了 TCP 这种面向流的协议无法避免的队首阻塞问题（参见 2.4 节“队首阻塞”）。类似地，同样的机制也让 SCTP 得以按配置交付：有序和乱序、可靠和部分可靠。

既然如此，为什么不直接在 IP 协议上运行 SCTP，然后通过它实现所有数据交换呢？这样做就不需要 UDP 了，而且也能解决将来通过 TCP 交付 HTTP 2.0 的问题（参见 12.3.5 节中的“丢包、高 RTT 连接和 HTTP 2.0 性能”）。事实上，SCTP 可以解决 HTTP 2.0 解决的大多数问题，也能让我们大幅度简化 HTTP 协议！

只可惜现有的路由器和 NAT 设备不能正确处理 SCTP，因而几乎不可能在公共互联网上将 SCTP 用作“裸传输协议”。WebRTC 这才在 UDP 和 DTLS 之上架设起 SCTP，而且在 WebRTC 客户端中，SCTP 是在“用户空间”中实现的。

在内部网等受控的环境下，SCTP 的性能很好。很多移动运营商都使用 SCTP 传输来自无线电发射塔的数据，直到穿过其核心网络传输到公共互联网。要了解这方面的更多信息，请参考这个链接：http://tools.ietf.org/html/draft-ietf-behave-sctpnat。

18.6 DataChannel

DataChannel 支持端到端的任意应用数据交换，就像 WebSocket 一样，但是端到端的，而且可以定义底层传输协议的交付属性。建立 RTCPeerConnection 连接之后，两端可以打开一或多个信道交换文本或二进制数据：

```
function handleChannel(chan) { ❶
  chan.onerror = function(error) { ... }
  chan.onclose = function() { ... }

  chan.onopen = function(evt) {
    chan.send("DataChannel connection established. Hello peer!")
  }

  chan.onmessage = function(msg) {
    if(msg.data instanceof Blob) {
      processBlob(msg.data);
    } else {
      processText(msg.data);
    }
  }
}

var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection(iceConfig);

var dc = pc.createDataChannel("namedChannel", {reliable: false}); ❷

... ❸

handleChannel(dc); ❹
pc.onDataChannel = handleChannel; ❺
```

❶ 在 DataChannel 对象上注册类似 WebSocket 的回调
❷ 以最合适的交付语义初始化新的 DataChannel
❸ 常规的 RTCPeerConnection 提议 / 应答代码
❹ 在本地初始化的 DataChannel 上注册回调
❺ 在远端初始化的 DataChannel 上注册回调

DataChannel API 有意照搬 WebSocket：每个信道都会触发同样的 onerror、onclose、onopen

和 onmessage 回调，而且每个信道也会提供同样的 binaryType、bufferedAmount 和 protocol 字段。

不过，由于 DataChannel 是端到端的，而且在更灵活的传输协议之上运行，因此它还具备一些 WebSocket 没有的功能。前面代码示例就展示了这其中一些最重要的差别：

- 与 WebSocket 构造函数不同，它需要传入 WebSocket 服务器的 URL，而 DataChannel 只是 RTCPeerConnection 对象的一个工厂方法；
- 与 WebSocket 不同，任何一端都可以初始新的 DataChannel 会话，会话建立后就会触发 onDataChannel 回调；
- 与 WebSocket 不同，它运行在可靠有序的 TCP 协议之上，而每个 DataChannel 都可以经过配置，实现自定义的交付和可靠性。

DataChannel 与 WebSocket API 的对比（表 18-2）

DataChannel API 是 WebSocket API 的超集，因此我们前面讨论过的关于 WebSocket 的一切，包括回调、标志、对处理文本和二进制数据的优化，以及子协议协商，都可以直接对应到 DataChannel API；参见 17.1 节“WebSocket API”。

表18-2：WebSocket与DataChannel

	WebSocket	DataChannel
加密	可配置	始终加密
可靠性	可靠	可配置
交付方式	有序	可配置
多路复用	不支持（有扩展）	支持
传输机制	面向消息	面向消息
二进制传输	支持	支持
UTF-8 传输	支持	支持
压缩	不支持（有扩展）	不支持

当然啦，WebSocket 与 DataChannel 最大的区别，还是底层传输协议不一样。WebSocket 运行在 TCP 之上，每条消息的交付都是可靠且有序的；而 DataChannel 运行在下列三个协议之上：

- UDP 提供端到端的连接；
- DTLS 对传输的数据加密；
- SCTP 支持多路复用、流量和拥塞控制及其他功能。

DataChannel 经过配置，可以提供与 WebSocket 相同的可靠性和有序交付语义。当然，更重要的是，DataChannel 真正的威力恰恰在于它不一定必须使用有序和可靠的语义！每个信道可以指定自己的交付和可靠性要求，而数据可以直接从一端传送到另一端。

18.6.1 设置与协商

无论传输的是什么类型的数据（音频、视频，还是应用数据），通信两端必须首先经过完整的提议 / 应答流程，协商要使用的协议和端口，成功完成连接检查（参见 18.4 节“建立端到端的连接”）。

事实上，正像我们现在已经知道的，传输媒体使用 SRTP，而 `DataChannel` 使用 SCTP 协议。因此，初始端在生成连接提议，或者另一端在生成应答时，它们两个必须特意在生成的 SDP 字符串中包含 SCTP 关联的参数：

```
(……省略的内容……)
m=application 1 DTLS/SCTP 5000 ❶
c=IN IP4 0.0.0.0 ❷
a=mid:data
a=fmtp:5000 protocol=webrtc-datachannel; streams=10 ❸
(……省略的内容……)
```

❶ 告知对方想使用 DTLS 之上的 SCTP
❷ 0.0.0.0 候选项表示使用增量 ICE
❸ SCTP 之上的 `DataChannel` 协议，最多 10 个并行流

跟以前一样，只要任意一端在生成会话的 SDP 描述前注册 `DataChannel`，`RTCPeerConnection` 就会负责生成 SDP 参数。实际上，应用通过明确设置禁用音频和视频传输的约束，可以建立只传输数据的端到端连接：

```
var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection(iceConfig);

var dc = pc.createDataChannel("namedChannel", {reliable: false}); ❶

var mediaConstraints = { ❷
  mandatory: {
      OfferToReceiveAudio: false,
      OfferToReceiveVideo: false
  }
};

pc.createOffer(function(offer) { ... }, null, mediaConstraints); ❸

...
```

❶ 通过 `RTCPeerConnection` 注册新的不可靠的 `DataChannel`
❷ 设置媒体约束，以禁用音频和视频传输
❸ 生成只传输数据的提议

协商确定了 SCTP 参数后，很快就可以交换应用数据。注意，前面看到的 SDP 片段中并未提到每个 `DataChannel` 的参数，比如协议、可靠性，或者有序或乱序标签。因此，在可以发送应用数据之前，初始的 WebRTC 客户端还要发送 `DATA_CHANNEL_OPEN` 消息（图 18-16），这个消息描述了数据类型、可靠性、要使用的应用协议，及信道的其他参数。

位	+0..7	+8..15	+16..23	+24..31
0	消息类型（0x3）	信道类型	优先级	
32	可靠性			
64	标签长度		协议长度	
...	标签			
...	协议			

图 18-16：`DATA_CHANNEL_OPEN` 消息初始化新信道

> `DATA_CHANNEL_OPEN` 消息与 HTTP 2.0 中的 `HEADERS` 帧类似：它显式地打开一个新流，随后可以立即发送数据帧；参见 12.4.1 节“发起新流”。要了解有关 DataChannel 协议的更多信息，请参考这个链接：http://tools.ietf.org/html/draft-jesup-rtcweb-data-protocol。

沟通完信道参数，两端就可以交换应用数据了。本质上，每个信道都作为一个独立的 SCTP 流发送数据，即所有信道都是在同一个 SCTP 关联之上多路复用出来的。这样就可以避免不同流之间的队首阻塞，在同一个 SCTP 关联上同时打开多个信道。

外部信道协商

`DataChannel` 也允许通过外部协商信道参数。在调用 `createDataChannel` 方法时，应用可以把 `negotiated` 参数设置为 `true`，这样就不会自动发送 `DATA_CHANNEL_OPEN` 消息。不过，这样一来，两端就必须指定相同的 `id` 参数，这个参数可以由浏览器自动生成：

```
signalingChannel.send({ ❶
  newchannel: true,
  label: "negotiated channel",
  options: {
    negotiated: true,
    id: 10, ❷
    reliable: true,
    ordered: true,
    protocol: "appProtocol-v3"
```

```
    }
  });

  signalingChannel.onmessage = function(msg) {
    if (msg.newchannel) { ❸
      dc = pc.createDataChannel(msg.label, msg.options);
    }
  }
```

❶ 通过发信通道向另一端发送信道配置

❷ 唯一的应用指定的信道 ID（整数）

❸ 用接收到的参数初始化新 DataChannel

实践中，如果通信端的数量不多，使用外部协商的性能优势并不明显。这时候，还是让 RTCPeerConnection 对象帮我们完成协商更好。然而，在通信端的数量很多的情况下，发信服务器可以生成相同的描述，同时将它们分发给所有通信端。此时，这种工作流就能派上用场。

18.6.2　配置消息次序和可靠性

DataChannel 可以让我们使用兼容 WebSocket 的 API 端到端地传输任意数据，就其本身而言，这绝对是独一无二而且十分强大的功能。可是，DataChannel 还支持一种灵活得多的传输方式，因此我们可以自定义每个信道的交付语义，以满足应用和要传输的数据类型的需要：

- DataChannel 可以有序或乱序交付消息；
- DataChannel 可以可靠或部分可靠地交付消息。

当然，要把信道配置成有序且可靠交付，那就与 TCP 无异了：与常规的 WebSocket 交付效果相同。然而，DataChannel 还为每个信道配置部分可靠性提供了两个选择：

- 通过重传实现部分可靠交付

  消息的重传次数由应用指定。

- 通过超时实现部分可靠交付

  消息的重传间隔（ms）由应用指定。

这两个选择都由 WebRTC 客户端实现，意味着应用所要做的，只是决定使用什么交付模型，然后对信道设置正确的参数即可。应用不管理计时器或重传计数器。下面我们再介绍一下配置选项（表 18-3）。

表18-3：DataChannel的可靠性和交付模型配置

	有序	可靠	部分可靠策略
有序 + 可靠	是	是	N/A
乱序 + 可靠	否	是	N/A
有序 + 部分可靠（重传）	是	部分	重传计数
乱序 + 部分可靠（重传）	否	部分	重传计数
有序 + 部分可靠（定时）	是	部分	超时（ms）
乱序 + 部分可靠（定时）	否	部分	超时（ms）

有序且可靠交付很好理解，就是 TCP。另一方面，乱序且可靠交付很有意思，它也是 TCP，但没有队首阻塞问题（参见 2.4 节“队首阻塞”）。

配置部分可靠的信道时，关键是要记住两个重传策略是互斥的。换句话说，应用可以指定超时重传，也可以指定计数重传，但不能同时指定两个策略；否则，就会报错。好了，下面我们来看一看用于配置信道的 JavaScript API 吧：

```
conf = {}; ❶
conf = { ordered: false };  ❷
conf = { ordered: true,  maxRetransmits: customNum };  ❸
conf = { ordered: false, maxRetransmits: customNum };  ❹
conf = { ordered: true,  maxRetransmitTime: customMs };  ❺
conf = { ordered: false, maxRetransmitTime: customMs };  ❻

conf = { ordered: false, maxRetransmits: 0 };  ❼

var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection(iceConfig);

...

var dc = pc.createDataChannel("namedChannel", conf);  ❽

if (dc.reliable) {
  ...
} else {
  ...
}
```

❶ 默认为有序可靠交付（TCP）
❷ 可靠乱序交付
❸ 有序、部分可靠，使用自定义的重传计数
❹ 乱序、部分可靠，使用自定义的重传计数
❺ 有序、部分可靠，使用自定义的超时重传
❻ 乱序、部分可靠，使用自定义的超时重传

❼ 乱序不可靠交付（UDP）
❽ 使用指定的配置初始化 DataChannel

> 初始化 DataChannel 之后，应用可以访问 maxRetransmits 和 maxRetransmitTime 这两个只读属性。同样，为方便起见，DataChannel 也提供了一个 reliable 属性，只要使用了部分可靠策略，这个属性就会返回 false。

每个 DataChannel 可以使用不同的自定义参数（可靠性及是否有序）来配置，每一端可以打开多个信道，这些信道都是通过对相同的 SCTP 关联多路复用实现的。结果，信道之间相互独立，可以分别用于传输不同类型的数据。比如，可靠有序的信道可用于聊天，部分可靠且乱序的交付可用于传输瞬时或低优先级的应用更新。

18.6.3 部分可靠交付与消息大小

要使用部分可靠的信道，需三思而后行。尤其是，应用必须密切关注消息大小：应用传输的数据可能很大，因此会被分段封装到多个分组中，而这很可能导致难以接受的结果。为说明这个问题，假设有以下场景。

- 两端协商好了一个乱序不可靠的 DataChannel。
  - 信道的 maxRetransmits 为 0，即纯粹是 UDP。
- 丢包率大约为 1%。
- 其中一端想要发送一个 120 KB 的大消息。

WebRTC 客户端将 SCTP 分组的最大传输单位设置为 1280 字节，这是针对 IPv6 分组推荐的 MTU。但我们也必须考虑到 IP、UDP、DTLS 和 SCTP 协议的开销，分别是 20~40 字节、8 字节、20~40 字节和 28 字节，加起来大约为 130 字节。这样每个分组就剩下大约 1150 字节给数据净荷。因此，120 KB 应用消息总共需要 107 个分组（120 × 1024/1150）。

现在还看不出有问题，但每个分组丢失的概率为 1%。如果我们通过不可靠信道发送全部 107 个分组，那么极有可能在中途会丢失其中一个！这会导致什么结果呢？即使仅仅丢失一个分组，整个消息也会被抛弃。

为解决这个问题，应用有两个选择：可以使用重传策略（计数或超时），或者减少消息大小。事实上，为了获得最佳结果，应该双管齐下。

- 在使用不可靠信道时，理想情况下，每个消息都应该可以封装到一个分组中。换句话说，消息应该小于 1150 字节。
- 如果消息不能封装到一个分组中，则需要使用重传策略，以提高交付消息的成功率。

端到端之间的丢包和延迟难以预测，会因当时的网络环境而变化。因此，对于重传次数，或者超时间隔，并没有唯一的最优答案。为了通过不可靠信道达成最佳结果，请尽量减少消息大小。

18.7 WebRTC使用场景及性能

实现低延迟、端到端的传输可不是件轻而易举的事，它涉及 NAT 穿越和连接检查、信号发送、安全加密、拥塞控制及大量其他必须考虑到的细节。WebRTC 为我们处理了上述所有这些，而且还不止这些。正因为如此，WebRTC 自问世起就被人们追捧为对 Web 平台有史以来最重要的补充。关键在于，WebRTC 不仅是实现了这些功能，而且保证了所有组件协同工作，为我们在浏览器中构建端到端应用提供了简单统一的 API。

不过，即便有了内置的服务，设计高效高性能的端到端应用仍然需要周密考虑和规划：端到端本身并不代表高性能。如果说有什么不同，那就是端到端之间的带宽和延迟越来越多变，对媒体传输的需求日益增长，以及不可靠交付的诸多特性反倒进一步增加了这种应用的构建难度。

18.7.1 音频、视频和数据流

端到端的音频和视频流是 WebRTC 最主要的使用场景之一：`getUserMedia` API 可以让应用获取媒体流，而内置的音频和视频引擎负责处理优化、错误恢复和流之间的同步。不过不要忘了，即便经过十分激进的优化和压缩，音频和视频交付依旧可能受到延迟和带宽的限制：

- 高清的流至少需要 1~2 Mbit/s 带宽，参见 18.2 节中的“音频（Opus）与视频（VP8）比特率”；
- 截至 2013 年第一季度，全球平均带宽只有 3.1 Mbit/s，参见表 1-2；
- 高清的流至少要求 3.5G+ 的连接，参见表 7-2。

好消息是，世界平均带宽一直在稳步增长：用户正在向宽带转移，而 3.5G+ 和 4G 网络的应用速度也在加快。可是，就算乐观估计，也只能说高清流在今天是可行的，还不能说有保证！类似地，延迟是一个老生常谈的问题，特别是对实时交付以及移动客户端，延迟的问题更加严重。4G 当然好，但 3G 网络要退出历史舞台也不是一朝一夕的事。

> 更严重的是，大多数 ISP 和移动运营商提供的连接并不对称：大多数用户的下行吞吐量明显高于上行吞吐量。事实上，10∶1 的比例不在少数，也就是说下载 10 Mbit/s，上传只有 1 Mbit/s。

最终结果就是，仅仅一条端到端的音频和视频流，就会占用大部分的用户带宽，而移动客户端就是更如此了。那如果是多方流呢？肯定需要对可用带宽有一个明确的把握：

- 移动客户端或许可以下载高清流（1 Mbit/s 以上），但囿于上行吞吐量，或许只能发送低品质流，即不同通信端接收和发送流的比特率不同；
- 音频和视频流可能需要与其他应用和数据传输服务共享带宽，比如多个 `DataChannel` 会话；
- 带宽和延迟一直在变，与连接类型（有线或无线）或者第几代网络无关，因此应用必须能适应这些变化。

好在 WebRTC 的音频和视频引擎会与底层的网络传输组件协同，去探测可用带宽，从而优化媒体流的交付。可是，`DataChannel` 还需要额外的应用逻辑：应用必须监控缓冲区中的数据量，随时准备按需作出调整。

> 在采集音频和视频流时，一定要将视频约束设置为与使用场景匹配，参见 18.2 节中的“通过 `getUserMedia` 获取音频和视频”。

18.7.2 多方通信架构

一个双向发送高清媒体流的端到端连接，很容易耗尽用户的带宽。为此，对于多方通信的场景，就更应该仔细考虑其架构了（图 18-17），特别是单个流如何汇聚，以及如何在端与端之间分发。

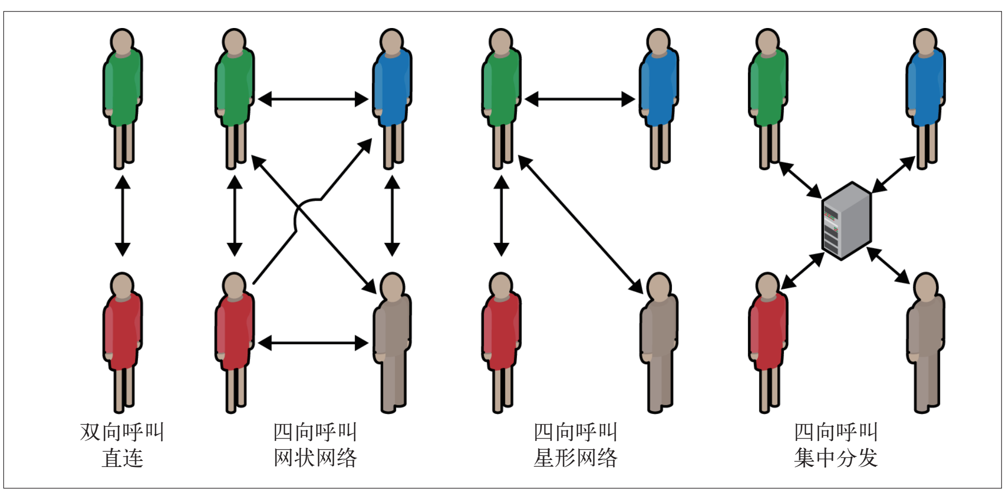


图 18-17：*N* 向呼叫的分布式架构

一对一连接最容易处理和部署：两端直连即可，不需要进一步优化。可是，如果对一个 N 方呼叫扩展这个策略，那么每一端都要分别连接到其他端（网状网络），即每一端有 $N-1$ 个连接，总共是 $N\times(N-1)$ 个连接！如果带宽有限，特别是上行速度更低，那么这种架构下只需少数参与端就会消耗掉用户的大多数带宽。

网状网络虽然容易建立，但对多方流交换而言往往效率很低。为解决这个问题，替代方案是使用“星形”拓扑，让每一端连接到一个“超级节点”，由它负责向连接各方分发流。这样，只有一个节点需要处理和分发 $N-1$ 个流，其他节点都与它直接对话。

超级节点可以是普通的参与端，也可以是一个为处理和分发实时数据而优化过的专用服务器。至于哪个策略更合适，还得看使用场景和应用。最简单的情况下，发起连接的一端可以作为超级节点——简单，但效果不可能太好。更好的办法则是选择可用吞吐量最大的通信端，但这样又需要额外的“挑选”和发信机制。

> 挑选条件和过程由应用决定，而这本身又是一个难题。WebRTC 没有为此提供任何辅助机制。

最后，超级节点可以是专用的，甚至还可以是第三方服务。WebRTC 能让我们实现端到端的通信，但这并不意味着不能考虑集中式架构！如果每一端都与代理服务器建立连接，那么既可以利用 WebRTC 提供的传输机制，又可以使用服务器提供的服务。

端到端优化即服务

很多现有的视频会议解决方案（比如谷歌的 Hangouts）都依赖“代理服务器”汇聚个别的媒体流，然后合成，再将优化的版本分发给连接各端。只交付一个流可以减少对每一端带宽和 CPU、GPU 资源的占用，因为每一端只看到一个流，而非 $N-1$ 个！

类似地，一台游戏服务器可以汇聚所有玩家的更新，通过筛选，只分发必要的更新。比如，不给处在视野之外的玩家发送更新，也不会影响其他玩家。

两端的流交换很简单，部署效率高，而多方架构则要求考虑很多，周密规划。WebRTC 很大程度上是为端到端直接通信服务的，因此也催生了很多其他服务，有商业的，也有开源的。这些服务反过来又会提升 WebRTC 应用的效率，丰富其功能。

18.7.3　基础设施及容量规划

除了规划和预测每一端的带宽需求，WebRTC 应用还需要某些集中式基础设施来发信、穿越 NAT 和防火墙、身份验证，以及其他服务。

WebRTC 把发信功能留给应用实现，这意味着应用最低限度也要实现两端的消息发送和接收逻辑。发信数据量因用户数量、协议、数据编码和更新频率而异。类似地，发信服务的延迟对“呼叫建立”时间（尤其是使用增量 ICE 的时候）及其他发信交换性能影响极大。

- 使用低延迟的传输方案，比如 WebSocket 或基于 XHR 的 SSE。
- 估计并提供足够的容量，以满足所有应用必需的信号发送速率。
- 可选地，建立连接后，各端可以切换到 `DataChannel` 来发信。这样可以转移中心服务器必须处理的发信流量，也会减少发信过程的延迟。

由于 NAT 和防火墙的广泛存在，大多数 WebRTC 应用都将需要 STUN 服务器来执行 IP 查找，从而建立端到端连接。好在，只有连接设置阶段才需要 STUN 服务器，但无论如何，它都必须采用 STUN 协议且随时可用，以处理必要的查询。

- 除非 WebRTC 应用只在同一内部网络中使用，否则始终都要在 RTCPeerConnection 对象中提供 STUN 服务器；不然，大多数连接会直接失败。
- 与发信服务器可以使用任何协议不同，STUN 服务器必须响应 STUN 请求。可以选择一台公共服务器，也可以自己配置；stund 就是一个流行的开源实现。

即便有了 STUN，8%~10% 的端到端连接也可能由于网络策略存在的各种问题而失败。比如，网络管理员可能会屏蔽网络中所有用户的 UDP 数据报（参见 3.2.3 节中的“现实中的 STUN 和 TURN”）。为此，要保证用户体验，应用可能还需要一台 TURN 服务器在端与端之间转发数据。

- 转发数据是一种次优选择：转发就会导致多余的网络跳跃，而在每个流占用 1 Mbit/s 带宽的情况下，任何服务都可能很快被耗尽带宽。所以，TURN 应该是没有办法的办法，实在不行才用，而且即便用，也要仔细对待容量规划问题。

多方服务可能需要集中式基础设施来辅助优化多个流的交付，从而成为 RTC 体验的一部分。很多时候，多方网关会扮演与 TURN 相同的角色，但这里的需求不同。话虽如此，但与 TURN 服务器充当简单的分组代理不同，这些“智能代理”在向各端转发最终输出前，可能要占用更多 CPU 和 GPU 资源来处理每一个流。

18.7.4　数据效率及压缩

WebRTC 音频和视频引擎会根据端与端之间的网络条件动态调整媒体流的比特率。应用可以设置和更新媒体约束（比如，视频的解析度、帧速率，等等），剩下的事交给引擎就好了——这一块很简单。

可惜的是，对于传输任意数据的 DataChannel 而言就没有那么简单了。与 WebSocket 类似，DataChannel API 可以接收二进制或 UTF-8 编码的应用数据，但它不能压缩数据：优化二进制净荷和压缩 UTF-8 数据是 WebRTC 应用的事。

此外，WebSocket 运行在可靠有序的传输协议之上，而 WebRTC 应用必须考虑 UDP、DTLS 和 SCTP 协议产生的额外开销，以及在部分可靠的连接上交付数据的各种问题（参见 18.6.3 节“部分可靠交付与消息大小”）。

> WebSocket 有一个协议扩展，支持对传输数据的自动压缩。然而，WebRTC 没有类似的扩展。因此，应用提供什么消息，它就传输什么消息。

18.8 性能检查表

端到端的架构对应用设计提出了独特的挑战。直接、一对一的通信相对简单，而参与端越多，问题就越复杂，至少对性能来说如此。最后，我们给出有助于提高端到端 WebRTC 应用性能的一些注意事项。

- 发信服务
  - 使用低延迟传输机制；
  - 提供足够的容量；
  - 建立连接后，考虑使用 DataChannel 发信。
- 防火墙和NAT穿越
  - 初始化 RTCPeerConnection 时提供 STUN 服务器；
  - 尽可能使用增量 ICE，虽然发信次数多，但建立连接速度快；
  - 提供 STUN 服务器，以备端到端连接失败后转发数据；
  - 预计并保证 TURN 转发时容量足够用。
- 数据分发
  - 对于大型多方通信，考虑使用超级节点或专用的中间设备；
  - 中间设备在转发数据前，考虑先对其进行优化或压缩。
- 数据效率
  - 对音频和视频流指定适当的媒体约束；
  - 优化通过 DataChannel 发送的二进制净荷；
  - 考虑压缩通过 DataChannel 发送的 UTF-8 数据；
  - 监控 DataChannel 缓冲数据的量，同时注意适应网络条件变化。

- 交付及可靠性
  - 使用乱序交付避免队首阻塞；
  - 如果使用有序交付，把消息大小控制到最小，以降低队首阻塞的影响；
  - 发送小消息（<1150 字节），以便将分段应用消息造成的丢包损失降至最低；
  - 对部分可靠交付，设置适当的重传次数和超时间隔；“正确的”设置取决于消息大小、应用数据类型和端与端之间的延迟。

关于封面

本书封面上的动物是马达加斯加鹞鹰（鹞属小鹰）。这种鹞鹰主要生活在科摩罗群岛（位于马达加斯加北面和非洲大陆东南面之间的印度洋中）和马达加斯加岛，由于生存环境缩小或遭受破坏，总量不断减少。最近发现这种鹞鹰的数量比原来想象得还要少，它们小群散居于岛上各处，每群大约 250~500 只成年个体。

在马达加斯加岛上的湿地附近，植被环绕的湖泊、沼泽、滨海湿地和水田，是这种鹞鹰最常出没的捕食场所。它们主要猎食小型无脊椎动物和昆虫，包括小型鸟、蛇、蜥蜴、鼠类和家禽。捕食家禽的习性（数量只占总捕食量的 1%），导致当地居民对它们大开杀戒。

每年的枯水季节（8 月末及 9 月）是这种鹞鹰的繁殖季节。雨季开始的时候，孵化（约 32~34 天）已经结束，接下来雏鹰的丰羽期大约为 42~45 天。然而，这种鹞鹰的繁殖率始终不高，平均每巢繁殖丰羽幼鹰 0.9 只，而只有四分之三的巢可以安然度过繁殖期。这么低的繁殖率缘于年复一年大范围的草原和沼泽烧荒（本地居民掏岛蛋及捅鸟窝也是一方面原因），目的是让大地长出新鲜的牧草和开荒，而且经常发生在鹞鹰繁殖的季节。鹞鹰繁衍需要安定的环境，而马达加斯加岛上的居民不断对土地的开发利用对此形成了威胁。

为保护这种鹞鹰，人们提出了一些建议，包括进一步调查摸清这种鹞鹰的具体数量，研究它们的种群动态，取得关于出巢率的准确信息，控制重点地带的烧荒行为——特别是在繁殖季节，以及围绕重点建巢区域划定和建立保护区。

本书封面的这张图片来自 *Histoire Naturelle, Ornithologie, Bernard Direxit*。本书英文版封面字体使用的是 Adobe ITC Garamond，正文字体使用的是 Adobe Minion Pro，标题字体使用的是 Adobe Myriad Condensed，代码字体使用的是 Dalton Maag 公司开发的 Ubuntu Mono。

Web Development/Networking

Web性能权威指南

怎么才能让Web应用速度快、效率高？本书为所有关心这个问题的人提供了必须知道的网络知识，既包括影响性能的最基本因素，也包括那些能让我们创造更强大Web应用的重要技术革新，比如HTTP 2.0、XHR的改进、服务器发送事件（SSE）、WebSocket和WebRTC等。

本书作者是世界顶尖的Web性能工程师，他在书中深入浅出地讲解并演示了针对TCP、UDP和TLS协议的性能优化最佳实践，以及面向无线和移动网络进行优化时的特殊要求。随后，他全面剖析了浏览器技术的几项重大革新，包括使用这些新技术时在性能方面需要的独到考量。革命性的HTTP 2.0、XHR客户端网络脚本、基于SSE及WebSocket的实时数据流，以及通过WebRTC实现P2P通信，对这些面向未来的重大浏览器技术，本书都从性能优化的角度给出了详尽的解读和分析。

- 基于TCP、UDP和TLS交付最佳性能
- 通过3G、4G移动网络实现最佳性能
- 开发速度快同时又节能的移动应用
- 突破HTTP 1.x及其他浏览器协议的性能瓶颈
- 考虑将来通过HTTP 2.0交付最佳性能
- 在浏览器中实现高效的实时数据流通信
- 通过实时WebRTC创建高效的端到端视频会议及低延迟应用

“所有关注Web性能的人都应该看这本书，它是这个领域公认的权威参考指南。”

——Mark Nottingham
IETF下一代HTTP工作组
（HTTPbis Working Group）主席

Ilya Grigorik
谷歌“Web加速”（Make The Web Fast）团队的性能工程师、开发大使。他每天的主要工作就是琢磨怎么让Web应用速度更快，总结并推广能够提升应用性能的最佳实践。

封面设计：Randy Comer，张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

O'REILLY®
oreilly.com.cn

ISBN 978-7-115-34910-1
9 787115 349101 >
ISBN 978-7-115-34910-1
定价：69.00元